

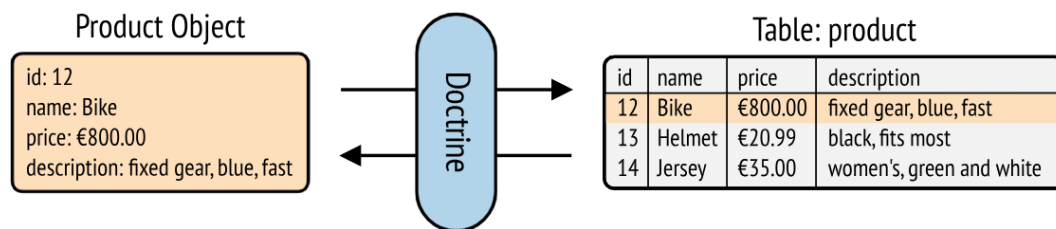
Actividad

Symfony ORM (Doctrine) y Bases de datos

1.	Introducción	2
2.	Actividad práctica	4
3.	Instalación de Doctrine	6
4.	Configuración de la base de datos	6
5.	Crear la base de datos con Doctrine	7
6.	Creación de una Entidad Doctrine.....	7
7.	Migraciones: Creación de las tablas/esquemas de la base de datos	9
8.	Migraciones y adición de más campos a las clases Entidad	11
9.	Persistir objetos en la base de datos	12
10.	Alcanzando objetos de la base de datos	15
11.	Usando la clase EntidadRepository para alcanzar objetos.....	15
12.	Actualización de un objeto.....	17
13.	Eliminación de un objeto.....	18
14.	Consulta de objetos: el repositorio	18
15.	Consultas con SQL	20

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** cuyo objetivo es proporcionar una herramienta potente para el tratamiento de la información, mediante **el mapeo de objetos a relaciones y viceversa**.



Un **ORM** (*Object-Relational Mapping* o *Mapeo Objeto-Relacional*) **es una técnica que permite interactuar con bases de datos relacionales utilizando objetos y programación orientada a objetos en lugar de realizar consultas SQL directamente**.

Los ORMs automatizan el proceso de traducir las estructuras y relaciones de una base de datos (tablas, columnas) a estructuras de objetos (clases, atributos) en un lenguaje de programación, y viceversa.

Propiedades de un ORM

1. Portabilidad:

- Los ORMs suelen soportar múltiples sistemas de bases de datos (MySQL, PostgreSQL, SQLite, etc.), facilitando el cambio de un motor de base de datos sin modificar el código.

2. Eficiencia:

- Optimizan las consultas mediante estrategias como lazy loading (carga diferida) o eager loading (carga anticipada) para reducir la carga en la base de datos.

3. Productividad:

- Ahorra tiempo y esfuerzo en la escritura de consultas SQL y manejo de datos relacionales, permitiendo a los desarrolladores centrarse en la lógica de negocio.

4. Seguridad:

- Previenen vulnerabilidades como SQL Injection al parametrizar automáticamente las consultas.

5. **Facilidad de mantenimiento:**

- Hace que el código sea más legible y organizado al tratar la base de datos como objetos del dominio en lugar de datos desconectados.

Desventajas del ORM

1. **Curva de aprendizaje:**

- Aprender a usar un ORM y sus configuraciones puede ser complicado para principiantes.

2. **Rendimiento:**

- En algunos casos, **las consultas generadas automáticamente pueden no estar optimizadas** en comparación con SQL escrito manualmente.

3. **Dependencia:**

- Cambiar de ORM o trabajar sin uno después de haberlo adoptado puede ser complejo.

4. **Comportamiento oculto:**

- Los desarrolladores deben entender cómo el ORM genera consultas para evitar problemas de rendimiento.

Ejemplos de ORMs :

- **PHP: Doctrine (utilizado en Symfony).Eloquent (Laravel).**
- **Java:** Hibernate, JPA (Java Persistence API).
- **Python:** SQLAlchemy,Django ORM.
- **JavaScript:** Sequelize,TypeORM.
- **.NET:** Entity Framework.

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 ORM para hacer persistencia de objetos en una base de datos relacional.

2.- Descarga el proyecto maqueta del aula virtual. Esta carpeta tiene lo necesario para comenzar con Symfony. Fijate en los archivos

Dockerfile

docker-compose.yml

NOTA: Antes de ejecutar adapta docker-compose.yml a tú ruta de proyecto

```
version: '3.8'
services:
  symfony-app:
    build:
      context: .      # Ubicación del Dockerfile
      dockerfile: Dockerfile
    container_name: symfony-linux
    volumes:
```

```
- << ponerruta>>:/var/www/symfony # Mapeo del proyecto
```

```
    working_dir: /var/www/symfony
    ports:
      - "8000:8000"
    environment:
      - COMPOSER_ALLOW_SUPERUSER=1
    command: tail -f /dev/null
    networks:
      - symfony-network
    depends_on:
      - mysql
  mysql:
```

```
image: mysql:8.0    # Imagen oficial de MySQL
container_name: mysql_sym
environment:
    MYSQL_ROOT_PASSWORD: 1234
    MYSQL_DATABASE: dawdb
volumes:
    - ./dataMySQL:/var/lib/mysql
ports:
    - "3306:3306"
networks:
    - symfony-network
```

```
networks:
    symfony-network:
        driver: bridge
```

NOTA: Fíjate que tu proyecto VisualStudio Code ira concetado a la carpeta SymfonyProyectos que en la ruta local del Docker estará en `var/www/symfony`

La salida será

```
✓Network symfonyproyectos_symfony-network    Created
✓Container mysql_sym                          Started
✓Container symfony-linux                      Started
```

Conecta al contenedor symfony-linux

```
docker exec -ti symfony-linux bash
```

Instala GIT

```
apt-get install git
```

Crea ahora un Proyecto denominado **labSymfony**

```
symfony new labSymfony1 --webapp
```

Ten paciencia tardará bastante

sitúate dentro de la carpeta del proyecto

```
cd labSymfony1
```

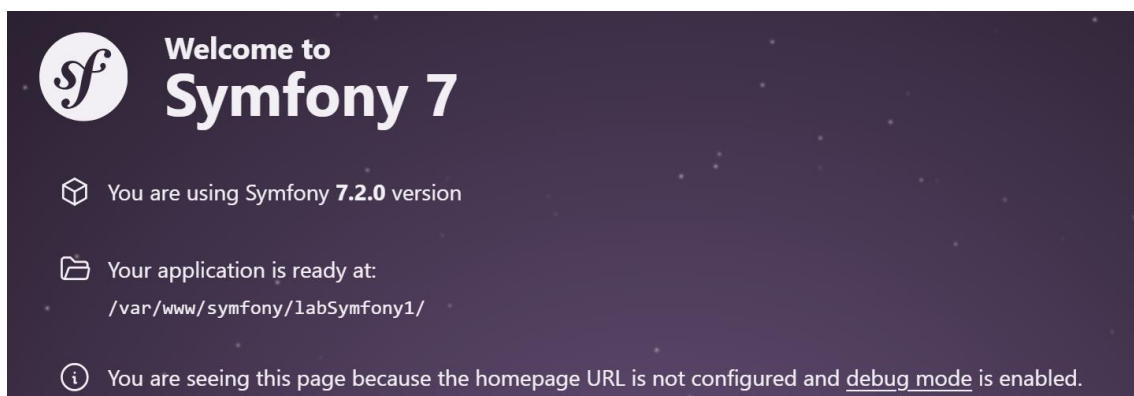
Estructura generada

```
labSymfony1/  
├─ config/  
├─ public/  
├─ src/  
├─ templates/  
├─ var/  
├─ vendor/  
├─ .env  
└─ composer.json
```

Arranca el servidor que Symfony tiene incorporado en el laboratorio

```
symfony server:start --allow-all-ip --no-tls --allow-http --port=8000
```

Comprueba que symfony está funcionando. Para ello accede mediante el navegador a <http://localhost:8000>



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
```

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="sqlite:///kernel.project_dir%/var/data.db"8
#
DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
#DATABASE_URL="mysql://root:mysql_sym:3306/dawdb?serverVersion=10.5.8-MariaDB"
# DATABASE_URL="mysql://root:1234@mysql_sym:3306/dawdb2?serverVersion=10.5.8-MariaDB"
# DATABASE_URL="mysql://root:1234@mysql_sym:3306/dawdb?serverVersion=8.0"
```

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 que **en nuestro caso es dawdb**.

Comprueba que se ha creado la base de datos correctamente. Para ello ejecuta accede a la consola de administración de mysql

```
docker exec -ti mysql_sym bash
mysql -u root -p
```

6. Creación de una Entidad Doctrine

Supongamos que está creando una aplicación que trabaja con tareas. 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

Sigue los siguientes pasos para crear una Entidad Doctrine denominada **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


```

<?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 (Tarea) en 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:

Ejecuta dentro del proyecto la siguiente instrucción

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

Si todo funcionó, deberías ver algo como esto:

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

[WARNING] You have 2 available migrations to execute.

Are you sure you wish to continue? (yes/no) [yes]:
> yes

created: migrations/Version20240105151055.php

Success!

Review the new migration then run it with php bin/console doctrine:migrations:migrate
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
```

Si abres (Version202401...php) este archivo contiene el SQL necesario para actualizar su base de datos.

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

```
<?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');
    }
}
```

Este comando ejecuta todos los archivos de migración que aún no se han ejecutado en la base de datos. Debes 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

```
daw@daw-VirtualBox: ~/my_project_directory$ php bin/console doctrine:migrations:migrate
WARNING! You are about to execute a migration in database "daw2" that could result in schema changes and data loss. Are you sure you wish to continue? (yes/no) [yes]:
> yes

[notice] Migrating up to DoctrineMigrations\Version20240105151055
[notice] finished in 160.2ms, used 12M memory, 3 migrations executed, 3 sql queries

[OK] Successfully migrated to version: DoctrineMigrations\Version20240105151055
```

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

¿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:

Añde un nuevo campo descripción a la Tarea

```
$ php bin/console make:entity

Class name of the entity to create or update
> Tarea

...(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

```
#[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

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');
    }

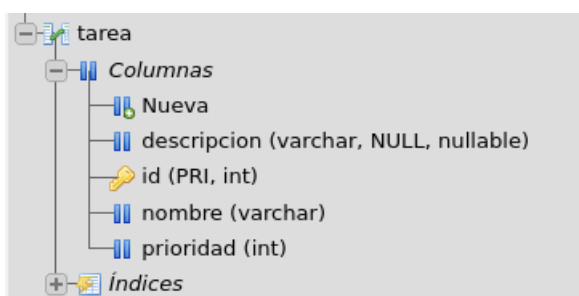
    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE product ADD description LONGTEXT NOT NULL');
        $this->addSql('ALTER TABLE tarea DROP descripcion');
    }
}

```

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 (TareaController)** para experimentar con esta función. Crea un controlador de entidades Tareas:

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

Como ves el comando crea dos tipos de componentes en tú proyecto

```
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 contenido creado del Controlador será el siguiente:

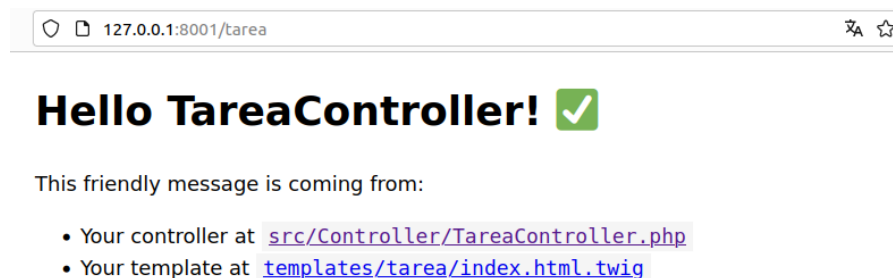
```
<?php

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:



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

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;

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 con id 5. Añade el siguiente método al `TareaController.php`

Nota : La etiqueta `#[Route('/tarea/{id}', name: 'tarea')]` en un controlador Symfony define una ruta HTTP específica para una acción (método) de ese controlador. Es parte de las *atribuciones* en PHP (introducidas en PHP 8) y es usada por Symfony para mapear URLs a controladores.

Rutas como `/tarea/{id}` son intuitivas porque:

- Indican el recurso:** En este caso, la ruta se asocia con el recurso `tarea`.
- Definen un identificador único:** `{id}` representa una instancia específica del recurso. Esto ayuda a entender que la URL `/tarea/42` hace referencia a la tarea con ID 42.

Este enfoque **sigue los principios de diseño de URL recomendados por la arquitectura RESTful**, donde las rutas reflejan la estructura de los datos y las acciones sobre los recursos.

```
#[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

11. Usando la clase EntidadRepository para alcanzar objetos

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

Añade

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

Y cambia el método show como sigue

```
#[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

NOTA :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

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)
            ->getQuery()->getResult();
    }
}

```

12. 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

```

#[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.

13. 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

```
#[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

14. Consulta de objetos: el repositorio

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

```
// from inside a controller
```

```
$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`:

```
/**@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()`**

```

#[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);
}

```

15. Consultas con SQL

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

```

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

```
#[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.

Anexo Resumen de comandos importantes:

Comando

```
symfony server:start --allow-all-ip --no-tls --allow-http --port=8000
```

El comando configura y ejecuta un servidor de desarrollo accesible desde cualquier dispositivo en tu red local, sin seguridad HTTPS, y en el puerto 8000.

Comando

```
symfony new labSymfony1 --webapp
```

Una manera rápida y estándar de crear aplicaciones Symfony completas, con todos los paquetes y herramientas necesarias preinstalados.

Es especialmente útil para principiantes o proyectos que necesitan una configuración web convencional.

Comando:

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

Crea un nuevo archivo de controlador:

- El archivo se genera dentro del directorio src/Controller.
- En este caso, el archivo será src/Controller/EjemploController.php.
- **Incluye una acción inicial:**

- Crea un método llamado `index()` dentro del controlador.
- Este método está asociado a una ruta predeterminada (`/ejemplo`) y devuelve una respuesta básica.
- **Genera una plantilla Twig:**
 - Si el paquete Twig está instalado, también crea automáticamente una plantilla para este controlador.
 - La plantilla estará en `templates/ejemplo/index.html.twig`.

Comando

- Es un comando proporcionado por el paquete Doctrine, el ORM (Object-Relational Mapping) usado en Symfony.
- Este comando **crea la base de datos** que se especifica en la configuración del proyecto Symfony.