Doctrine ORM

El framework symfony no incluye ningún componente para trabajar con bases de datos.
Sin embargo proporciona integración con una librería llamada Doctrine.

NOTA:
Doctrine ORM está totalmente desacoplado de Symfony y no es obligatorio utilizarlo.
Permite mapear objetos a una base de datos relacional (como MySQL, PostgreSQL o Microsoft SQL).

Si lo que se quiere es ejecutar consultas en bruto a la base de datos, se puede utilizar Doctrine DBAL

https://symfony.com/doc/current/doctrine/dbal.html

También se pueden utilizar bases de datos no relacionales como MongoDB.

http://symfony.com/doc/master/bundles/DoctrineMongoDBBundle/index.htm

Para utilizar Doctrine hay que instalarlo, junto con el bundle MakerBundle, que tiene las utilidades para generar código:

> *composer require doctrine*

> *composer require maker --dev*

Configurar el acceso a la base de datos

La información de conexión de base de datos está almacenada en una variable de entorno llamada **DATABASE_URL**:

```
# .env

# customize this line!
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name"

# to use sqlite:
# DATABASE_URL="sqlite:///%kernel.project_dir%/var/app.db"
```

Crear la base de datos

Podemos crear la base de datos a mano, o utilizar la consola de symfony:

> *bin/console doctrine:database:create*

Crear las entidades

Aquí igual: podemos crear a mano las entidades, o utilizar la consola de symfony.
Una entidad no es más que una clase decorada con decoradores de Doctrine.

> *bin/console make:entity*

Las entidades se guardan generalmente en el directorio *src/Entity*.

Podéis ver una lista de todos los comandos utilizando el siguiente comando:

> *php bin/console list doctrine*

Validar las entidades

Otro comando interesante es *doctrine:schema:validate* que valida las entidades.

> *bin/console doctrine:schema:validate*

Operaciones de INSERT, SELECT, UPDATE y DELETE

Persistir objetos en la base de datos (INSERT)

Utilizamos el objeto EntityManager para persistir objetos en la base de datos.

```
public function createAction()
{
    // Podemos obtener el EntityManager a través de $this->getDoctrine()
    // o con inyección de dependencias: createAction(EntityManagerInterface $em)
    $em = $this->getDoctrine()->getManager();

    $grado = new Grado();
    $grado->setNombre('Ingeniería de montes');

    // Informamos a Doctrine de que queremos guardar el Grado (todavía no se ejecuta ninguna query)
    $em->persist($grado);

    // Para ejecutar las queries pendientes, se utiliza flush().
    $em->flush();

    return new Response('El id del nuevo grado creado es '.$grado->getId());
```

```
}
```

Si tenemos definidas varias conexiones, podemos instanciar un objeto EntityManager para cada una de las conexiones.

```
$doctrine = $this->getDoctrine();
$em = $doctrine->getManager();
$em2 = $doctrine->getManager('other_connection');
```

Configurar varias conexiones es muy sencillo:

```
# app/config/config.yml
doctrine:
    dbal:
        default_connection:   default
        connections:
            # A collection of different named connections
(e.g. default, conn2, etc)
            default:
                dbname:               bd1
                host:                 10.0.1.6
                port:                 ~
                user:                 root
                password:             ~
                charset:              ~
                path:                 ~
```

```
                memory:                 ~
            other_connection:
                dbname:                 bd2
                host:                   10.0.1.7
                port:                   ~
                user:                   root
                password:               ~
                charset:                ~
                path:                   ~
                memory:                 ~
```

Recuperar objetos de la base de datos (SELECT)

```
public function showAction($id)
{
    $grado = $this->getDoctrine()
        ->getRepository(Grado::class)
        ->find($id);

    if (!$grado) {
        throw $this->createNotFoundException(
            'No se ha encontrado ningún grado con el id '.
$id
        );
    }
```

```
}
```

Las clases Repository

La clase Repository nos ofrece automáticamente varios métodos muy útiles para obtener registros de la base de datos:

- find
- findOneByXXX
- findByXXX
- findAll
- findOneBy
- findBy

```
$repository = $this->getDoctrine()->getRepository(Grado::class);

// Obtener un grado buscando por su primary key (normalmente "id")
$grado = $repository->find($gradoId);

// Métodos dinámicos para obtener un grado buscando por el valor de una columna
$grado = $repository->findOneById($gradoId);
$grado = $repository->findOneByNombre('Ingeniería de montes');

// Métodos dinámicos para obtener un array de objetos grad
```

```
o buscando por el valor de una columna
$grados = $repository->findByNombre('Ingeniería de montes'
);

// Obtener todos los grados
$grados = $repository->findAll();
```

```
$repository = $this->getDoctrine()->getRepository(Asignatu
ra::class);

// query for a single product matching the given name and
price
$product = $repository->findOneBy(
    array('nombre' => 'montes', 'credects' => 6)
);

// query for multiple products matching the given name, or
dered by price
$products = $repository->findBy(
    array('nombre' => 'matematicas'),
    array('credects' => 'ASC')
);
```

Estos métodos realizan búsquedas exactas (con el operador '='). Si queremos realizar búsquedas con el operador LIKE, tenemos que recurrir al lenguaje DQL o al objeto QueryBuilder que veremos más adelante.

La barra de depuración

Podemos consultar en la barra de depuración todas las SQLs ejecutadas, así como el tiempo y la memoria consumidos con cada petición.

Haciendo click en el icono de la base de datos vemos toda la información detallada.

Editar un objeto (UPDATE)

Editar un registro es igual de fácil que crearlo, simplemente en vez de hacer un new, partimos de un registro ya existente obtenido de la base de datos.

```
{
    $grado = $this->getDoctrine()->getRepository(Grado::cl
ass)->find($id);

    $grado->setNombre('otro nombre');

    $em->persist($grado);
    $em->flush();
    ...
}
```

Eliminar un objeto (DELETE)

Para eliminar un registro utilizamos el método remove() del EntityManager.

```
public function showAction($id, EntityManagerInterface $em
)
{
    $grado = $this->getDoctrine()->getRepository(Grado::cl
ass)->find($id);

    $em->remove($grado);
    $em->flush();
    ...
}
```

Relaciones entre entidades

https://symfony.com/doc/current/doctrine/associations.html

Las entidades Asignatura y Grado están relacionadas. Un asignatura pertenece a un grado y un grado tiene muchas asignaturas.Desde la perspectiva de la entidad Asignatura, es una relación *many-to-one*.

Desde la perspectiva de la entidad Grado, es una relación *one-to-many*.

La naturaleza de la relación determina qué metadatos de mapeo se van a utilizar.
También determina qué entidad contendrá una referencia a la otra entidad

Para relacionar las entidades Asignatura y Grado, simplemente creamos una propiedad
grado en la entidad Asignatura con las anotaciones que vemos a continuación:

```
class Asignatura
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="Grado", inversedBy="asignaturas")
     * @ORM\JoinColumn(name="grado_id", referencedColumnName="id")
     */
    private $grado;
}
```

This many-to-one mapping is critical. It tells Doctrine to use the

category_id column on the product table to relate each record in that table with a record in the category table.

Next, since a single Category object will relate to many Product objects, a products property can be added to the Category class to hold those associated objects.

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

class Grado
{
    // ...

    /**
     * @ORM\OneToMany(targetEntity="Asignatura", mappedBy="grado")
     */
    private $asignaturas;

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

La asociación many-to-one es obligatoria, pero la one-to-may es opcional.

El código en el constructor es importante. En lugar de ser instanciado como un array tradicional, la propiedad $asignaturas debe ser de un tipo que implemente la interface *DoctrineCollection*. El objeto ArrayCollection es de este tipo.

El objeto *ArrayCollection* parece y se comporta casi exactamente como un array por lo que todas las operaciones válidas sobre arrays, serán válidas sobre ArrayCollection.

Ya solamente queda crear los getters y setters correspondientes.

Si ahora actualizamos el schema, se generarán las relaciones en la base de datos

> *bin/console doctrine:schema:update --force*

Guardando entidades relacionadas

```
use AppBundle\Entity\Category;
use AppBundle\Entity\Product;
```

```
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    public function createProductAction()
    {
        $category = new Category();
        $category->setName('Computer Peripherals');

        $product = new Product();
        $product->setName('Keyboard');
        $product->setPrice(19.99);
        $product->setDescription('Ergonomic and stylish!')
;

        // relate this product to the category
        $product->setCategory($category);

        $em = $this->getDoctrine()->getManager();
        $em->persist($category);
        $em->persist($product);
        $em->flush();

        return new Response(
            'Saved new product with id: '.$product->getId(
)
            .' and new category with id: '.$category->getI
d()
```

```
        );
    }
}
```

Navegar entre entidades relacionadas

```
$product = $this->getDoctrine()
        ->getRepository(Product::class)
        ->find($productId);

    $categoryName = $product->getCategory()->getName();
```

También tenemos un método get en la otra entidad.

```
$category = $this->getDoctrine()
        ->getRepository(Category::class)
        ->find($categoryId);

    $products = $category->getProducts();
```

Para más tipos de asociaciones entre entidades hay que acudir a la documentación oficial de doctrine.
http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html

- @ManyToOne (unidireccional)
- @OneToOne (unidireccional / bidireccional / autorreferenciado)
- @OneToMany (bidirectional / unidireccional con join table / autorreferenciado)
- @ManyToMany (unidireccional / bidireccional / autorreferenciado)

Buenas Prácticas: Las consultas sql (dql se deben realizar en la clase Repository

Configuración de doctrine

Doctrine es altamente configurable. Es esta página se pueden consultar todas las opciones de configuración

https://symfony.com/doc/current/reference/configuration/doctrine.html

Extensiones de doctrine

Para ciertas estructuras y comportamientos habituales de tablas, hay desarrolladores que programan extensiones de doctrine.

https://symfony.com/doc/current/doctrine/common_extensions.html
https://github.com/Atlantic18/DoctrineExtensions

Documentación general

https://symfony.com/doc/current/doctrine.html

Relaciones en Doctrine

Los tipos de relaciones que soporta Doctrine son los siguientes:

- Many-To-One, Unidireccional

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToOne(targetEntity="Address")
     * @JoinColumn(name="address_id", referencedColumnName
="id")
     */
    private $address;
}

/** @Entity */
class Address
{
    // ...
}
```

- One-To-One, Unidireccional

```php
<?php
/** @Entity */
class Product
{
    // ...

    /**
     * One Product has One Shipment.
     * @OneToOne(targetEntity="Shipment")
     * @JoinColumn(name="shipment_id", referencedColumnNam
e="id")
     */
    private $shipment;

    // ...
}

/** @Entity */
class Shipment
{
    // ...
}
```

- One-To-One,

```php
<?php
/** @Entity */
class Customer
```

```
{
    // ...

    /**
     * One Customer has One Cart.
     * @OneToOne(targetEntity="Cart", mappedBy="customer")
     */
    private $cart;

    // ...
}

/** @Entity */
class Cart
{
    // ...

    /**
     * One Cart has One Customer.
     * @OneToOne(targetEntity="Customer", inversedBy="cart
")
     * @JoinColumn(name="customer_id", referencedColumnNam
e="id")
     */
    private $customer;

    // ...
}
```

- One-To-One, Auto-

```
<?php
/** @Entity */
class Student
{
    // ...

    /**
     * One Student has One Student.
     * @OneToOne(targetEntity="Student")
     * @JoinColumn(name="mentor_id", referencedColumnName="id")
     */
    private $mentor;

    // ...
}
```

- One-To-Many, Bidireccional

```
<?php
use Doctrine\Common\Collections\ArrayCollection;

/** @Entity */
class Product
{
```

```
    // ...
    /**
     * One Product has Many Features.
     * @OneToMany(targetEntity="Feature", mappedBy="produc
t")
     */
    private $features;
    // ...

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

/** @Entity */
class Feature
{
    // ...
    /**
     * Many Features have One Product.
     * @ManyToOne(targetEntity="Product", inversedBy="feat
ures")
     * @JoinColumn(name="product_id", referencedColumnName
="id")
     */
    private $product;
    // ...
}
```

- One-To-Many, Auto-referenciado

```
<?php
/** @Entity */
class Category
{
    // ...
    /**
     * One Category has Many Categories.
     * @OneToMany(targetEntity="Category", mappedBy="paren
t")
     */
    private $children;

    /**
     * Many Categories have One Category.
     * @ManyToOne(targetEntity="Category", inversedBy="chi
ldren")
     * @JoinColumn(name="parent_id", referencedColumnName=
"id")
     */
    private $parent;
    // ...

    public function __construct() {
        $this->children = new \Doctrine\Common\Collections
\ArrayCollection();
```

```
    }
}
```

- Many-To-Many, Unidireccional

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * Many Users have Many Groups.
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="users_groups",
     *      joinColumns={@JoinColumn(name="user_id", refer
encedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="group_id
", referencedColumnName="id")}
     *      )
     */
    private $groups;

    // ...

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\A
rrayCollection();
```

```
    }
}

/** @Entity */
class Group
{
    // ...
}
```

- Many-To-Many, Bidireccional

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * Many Users have Many Groups.
     * @ManyToMany(targetEntity="Group", inversedBy="users
")
     * @JoinTable(name="users_groups")
     */
    private $groups;

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\A
rrayCollection();
```

```
    }

    // ...
}

/** @Entity */
class Group
{
    // ...
    /**
     * Many Groups have Many Users.
     * @ManyToMany(targetEntity="User", mappedBy="groups")
     */
    private $users;

    public function __construct() {
        $this->users = new \Doctrine\Common\Collections\Ar
rayCollection();
    }

    // ...
}
```

- Many-To-Many, Auto-referenciado

```
<?php
/** @Entity */
class User
```

```
{
    // ...

    /**
     * Many Users have Many Users.
     * @ManyToMany(targetEntity="User", mappedBy="myFriend
s")
     */
    private $friendsWithMe;

    /**
     * Many Users have many Users.
     * @ManyToMany(targetEntity="User", inversedBy="friend
sWithMe")
     * @JoinTable(name="friends",
     *      joinColumns={@JoinColumn(name="user_id", refer
encedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="friend_u
ser_id", referencedColumnName="id")}
     *      )
     */
    private $myFriends;

    public function __construct() {
        $this->friendsWithMe = new \Doctrine\Common\Collec
tions\ArrayCollection();
        $this->myFriends = new \Doctrine\Common\Collection
s\ArrayCollection();
```

```
    }

    // ...
}
```

Transacciones

Transacción implícita o automática

```
$user = new User;
$user->setName('George');
$em->persist($user);
$em->flush();
```

Transacción explícita o manual

```
$em->getConnection()->beginTransaction(); // suspend auto-commit
try {
    //... do some work
    $user = new User;
    $user->setName('George');
    $em->persist($user);
    $em->flush();
    $em->getConnection()->commit();
} catch (Exception $e) {
    $em->getConnection()->rollBack();
    throw $e;
}
```

O bien de esta forma equivalente:

```php
<?php
$em->transactional(function($em) {
    //... do some work
    $user = new User;
    $user->setName('George');
    $em->persist($user);
});
```

El lenguaje DQL (Doctrine Query Language)

Si queremos ahorrar consultas, o realizar consultas complejas, podemos utilizar el leguaje DQL.

```
class AlumnoRepository extends \Doctrine\ORM\EntityReposit
ory
{
  public function findWithNotas($id)
  {
      $query = $this->getEntityManager()
          ->createQuery(
          'SELECT a, n, g, asig FROM App:Alumno a
          JOIN a.notas n
          JOIN a.grado g
          JOIN n.asignatura asig
          WHERE a.id = :id'
      )->setParameter('id', $id);

      try {
          return $query->getSingleResult();
      } catch (\Doctrine\ORM\NoResultException $e) {
          return null;
      }
  }
```

```
}
```

En el ejemplo anterior, en una única consulta SQL obtenemos el alumno, su grado, sus notas y las asignaturas de sus notas.

Se puede utilizar DQL para consultas de tipo SELECT, UPDATE y DELETE.

DQL no permite hacer INSERTS. Los inserts se deben realizar mediante el método persist() el EntityManager.

En las consultas de tipo SELECT con DQL, Doctrine hidrata los resultados en las Entidades correspondientes.

SELECT

Cuando trabajamos con DQL, no trabajamos con tablas y campos de las tablas sino con entidades y con propiedades de las entidades.

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\U
ser u WHERE u.age > 20');
$users = $query->getResult();
```

Formato de la respuesta

```
SELECT u, p, n FROM Users u...
```

En este caso Doctrine devuelve un array de objetos Users, con las entidades relacionadas p y n ya hidratadas porque forman parte del SELECT.

```
SELECT u.name, u.address FROM Users u...
```

En este caso Doctrine devuelve un array de arrays.

```
SELECT u, p.quantity FROM Users u...
```

En este caso Doctrine devuelve un array con una mezcla de objetos y valores.

JOINS

Los joins se realizan a través de las propiedades de las entidades

```
<?php
$query = $em->createQuery("SELECT u FROM User u JOIN u.add
ress a WHERE a.city = 'Berlin'");
$users = $query->getResult();
```

PARÁMETROS

Se pueden indicar parámetros con el símbolo '?' o el símbolo ':'

```php
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE
 u.id = ?1');
$query->setParameter(1, 321);
$users = $query->getResult();
```

```php
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE
 u.username = :name');
$query->setParameter('name', 'Bob');
```

```php
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE
 (u.username = :name OR u.username = :name2) AND u.id = :i
d');
$query->setParameters(array(
    'name' => 'Bob',
    'name2' => 'Alice',
    'id' => 321,
));
$users = $query->getResult();
```

FUNCIONES

En las clausulas SELECT, WHERE y HAVING se pueden utilizar las siguientes funciones de DQL:

- IDENTITY(single_association_path_expression [, fieldMapping]) - Devuelve la columna foreign key de la asociación
- ABS(arithmetic_expression)
- CONCAT(str1, str2)
- CURRENT_DATE()
- CURRENT_TIME()
- CURRENT_TIMESTAMP()
- LENGTH(str)
- LOCATE(needle, haystack [, offset]) - Busca la primera ocurrencia de un substring en un string.
- LOWER(str)
- MOD(a, b)
- SIZE(collection) - Devuelve el número de elementos de la colección
- SQRT(q) - Raíz cuadrada
- SUBSTRING(str, start [, length])
- TRIM([LEADING \ BOTH] ['trchar' FROM] str)
- UPPER(str)
- DATE_ADD(date, days, unit)
- DATE_SUB(date, days, unit)
- DATE_DIFF(date1, date2)

Se pueden utilizar operadores matemáticos

```
SELECT person.salary * 1.5 FROM CompanyPerson person WHERE
 person.salary < 100000
```

En las clausulas SELECT y GROUP BY se pueden utilizar las siguientes funciones de agregado:

- AVG
- COUNT
- MIN
- MAX
- SUM

Otras expresiones de SQL soportadas por DQL:

- ALL/ANY/SOME
- BETWEEN a AND b y NOT BETWEEN a AND b
- IN (x1, x2, …) y NOT IN (x1, x2, …)
- LIKE … y NOT LIKE …
- IS NULL y IS NOT NULL
- EXISTS y NOT EXISTS

El objeto QueryBuilder

Doctrine tiene también un constructor de queries llamado QueryBuilder, que facilita la construcción de sentencias DQL.

```
// src/Repository/ProductoRepository.php
namespace App\Repository;

use Doctrine\ORM\EntityRepository;

class ProductoRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        $qb = $this->createQueryBuilder('p')
                        ->where('p.price > :price')
                        ->setParameter('price', '19.99')
                        ->orderBy('p.price', 'ASC')
                        ->getQuery();

        return $qb->getResult();
    }
}
```

Parámetros

Se pueden indicar parámetros con el símbolo '?' o el símbolo ':'

```
$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = ?1')
   ->orderBy('u.name', 'ASC')
   ->setParameter(1, 100);
```

```
$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = :identifier')
   ->orderBy('u.name', 'ASC')
   ->setParameter('identifier', 100);
```

Existe un método para establecer varios parámetros de una sola vez:

```
$qb->setParameters(array(1 => 'value for ?1', 2 => 'value
for ?2'));
```

Tipos de respuesta

```
$result = $query->getResult();
$single = $query->getSingleResult();
$array = $query->getArrayResult();
$scalar = $query->getScalarResult();
$singleScalar = $query->getSingleScalarResult();
```

Métodos

La lista completa de métodos del objeto QueryBuilder es la siguiente:

```
<?php
class QueryBuilder
{
    // Example - $qb->select('u')
    // Example - $qb->select(array('u', 'p'))
    // Example - $qb->select($qb->expr()->select('u', 'p')
)
    public function select($select = null);

    // addSelect does not override previous calls to selec
t
    //
    // Example - $qb->select('u');
    //               ->addSelect('p.area_code');
    public function addSelect($select = null);

    // Example - $qb->delete('User', 'u')
    public function delete($delete = null, $alias = null);

    // Example - $qb->update('Group', 'g')
    public function update($update = null, $alias = null);

    // Example - $qb->set('u.firstName', $qb->expr()->lite
```

```
ral('Arnold'))
    // Example - $qb->set('u.numChilds', 'u.numChilds + ?1
')
    // Example - $qb->set('u.numChilds', $qb->expr()->sum(
'u.numChilds', '?1'))
    public function set($key, $value);

    // Example - $qb->from('Phonenumber', 'p')
    // Example - $qb->from('Phonenumber', 'p', 'p.id')
    public function from($from, $alias, $indexBy = null);

    // Example - $qb->join('u.Group', 'g', Expr\Join::WITH
, $qb->expr()->eq('u.status_id', '?1'))
    // Example - $qb->join('u.Group', 'g', 'WITH', 'u.stat
us = ?1')
    // Example - $qb->join('u.Group', 'g', 'WITH', 'u.stat
us = ?1', 'g.id')
    public function join($join, $alias, $conditionType = n
ull, $condition = null, $indexBy = null);

    // Example - $qb->innerJoin('u.Group', 'g', Expr\Join:
:WITH, $qb->expr()->eq('u.status_id', '?1'))
    // Example - $qb->innerJoin('u.Group', 'g', 'WITH', 'u
.status = ?1')
    // Example - $qb->innerJoin('u.Group', 'g', 'WITH', 'u
.status = ?1', 'g.id')
    public function innerJoin($join, $alias, $conditionTyp
e = null, $condition = null, $indexBy = null);
```

```
    // Example - $qb->leftJoin('u.Phonenumbers', 'p', Expr
\Join::WITH, $qb->expr()->eq('p.area_code', 55))
    // Example - $qb->leftJoin('u.Phonenumbers', 'p', 'WIT
H', 'p.area_code = 55')
    // Example - $qb->leftJoin('u.Phonenumbers', 'p', 'WIT
H', 'p.area_code = 55', 'p.id')
    public function leftJoin($join, $alias, $conditionType
 = null, $condition = null, $indexBy = null);

    // NOTE: ->where() overrides all previously set condit
ions
    //
    // Example - $qb->where('u.firstName = ?1', $qb->expr(
)->eq('u.surname', '?2'))
    // Example - $qb->where($qb->expr()->andX($qb->expr()-
>eq('u.firstName', '?1'), $qb->expr()->eq('u.surname', '?2
')))
    // Example - $qb->where('u.firstName = ?1 AND u.surnam
e = ?2')
    public function where($where);

    // NOTE: ->andWhere() can be used directly, without an
y ->where() before
    //
    // Example - $qb->andWhere($qb->expr()->orX($qb->expr(
)->lte('u.age', 40), 'u.numChild = 0'))
    public function andWhere($where);
```

```
    // Example - $qb->orWhere($qb->expr()->between('u.id',
 1, 10));
    public function orWhere($where);

    // NOTE: -> groupBy() overrides all previously set gro
uping conditions
    //
    // Example - $qb->groupBy('u.id')
    public function groupBy($groupBy);

    // Example - $qb->addGroupBy('g.name')
    public function addGroupBy($groupBy);

    // NOTE: -> having() overrides all previously set havi
ng conditions
    //
    // Example - $qb->having('u.salary >= ?1')
    // Example - $qb->having($qb->expr()->gte('u.salary',
'?1'))
    public function having($having);

    // Example - $qb->andHaving($qb->expr()->gt($qb->expr(
)->count('u.numChild'), 0))
    public function andHaving($having);

    // Example - $qb->orHaving($qb->expr()->lte('g.manager
Level', '100'))
```

```
    public function orHaving($having);

    // NOTE: -> orderBy() overrides all previously set ord
ering conditions
    //
    // Example - $qb->orderBy('u.surname', 'DESC')
    public function orderBy($sort, $order = null);

    // Example - $qb->addOrderBy('u.firstName')
    public function addOrderBy($sort, $order = null); // D
efault $order = 'ASC'
}
```

Limitar los resultados

```
<?php
// $qb instanceof QueryBuilder
$offset = (int)$_GET['offset'];
$limit = (int)$_GET['limit'];

$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('orderBy', 'u.name ASC')
   ->setFirstResult( $offset )
   ->setMaxResults( $limit );
```

La clase Expr

```php
<?php
// $qb instanceof QueryBuilder

// example8: QueryBuilder port of:
// "SELECT u FROM User u WHERE u.id = ? OR u.nickname LIKE
 ? ORDER BY u.name ASC" using Expr class
$qb->add('select', new Expr\Select(array('u')))
    ->add('from', new Expr\From('User', 'u'))
    ->add('where', $qb->expr()->orX(
        $qb->expr()->eq('u.id', '?1'),
        $qb->expr()->like('u.nickname', '?2')
    ))
    ->add('orderBy', new Expr\OrderBy('u.name', 'ASC'));
```

La lista completa de métodos de la clase Expr es la siguiente:

```php
<?php
class Expr
{
    /** Conditional objects **/

    // Example - $qb->expr()->andX($cond1 [, $condN])->add
(...)->...
    public function andX($x = null); // Returns Expr\AndX
instance
```

```
    // Example - $qb->expr()->orX($cond1 [, $condN])->add(
...)->...
    public function orX($x = null); // Returns Expr\OrX in
stance

    /** Comparison objects **/

    // Example - $qb->expr()->eq('u.id', '?1') => u.id = ?
1
    public function eq($x, $y); // Returns Expr\Comparison
 instance

    // Example - $qb->expr()->neq('u.id', '?1') => u.id <>
 ?1
    public function neq($x, $y); // Returns Expr\Compariso
n instance

    // Example - $qb->expr()->lt('u.id', '?1') => u.id < ?
1
    public function lt($x, $y); // Returns Expr\Comparison
 instance

    // Example - $qb->expr()->lte('u.id', '?1') => u.id <=
 ?1
    public function lte($x, $y); // Returns Expr\Compariso
n instance
```

```
    // Example - $qb->expr()->gt('u.id', '?1') => u.id > ?
1
    public function gt($x, $y); // Returns Expr\Comparison
 instance

    // Example - $qb->expr()->gte('u.id', '?1') => u.id >=
 ?1
    public function gte($x, $y); // Returns Expr\Compariso
n instance

    // Example - $qb->expr()->isNull('u.id') => u.id IS NU
LL
    public function isNull($x); // Returns string

    // Example - $qb->expr()->isNotNull('u.id') => u.id IS
 NOT NULL
    public function isNotNull($x); // Returns string

    /** Arithmetic objects **/

    // Example - $qb->expr()->prod('u.id', '2') => u.id *
2
    public function prod($x, $y); // Returns Expr\Math ins
tance

    // Example - $qb->expr()->diff('u.id', '2') => u.id -
2
    public function diff($x, $y); // Returns Expr\Math ins
```

```
tance

    // Example - $qb->expr()->sum('u.id', '2') => u.id + 2
    public function sum($x, $y); // Returns Expr\Math inst
ance

    // Example - $qb->expr()->quot('u.id', '2') => u.id /
2
    public function quot($x, $y); // Returns Expr\Math ins
tance

    /** Pseudo-function objects **/

    // Example - $qb->expr()->exists($qb2->getDql())
    public function exists($subquery); // Returns Expr\Fun
c instance

    // Example - $qb->expr()->all($qb2->getDql())
    public function all($subquery); // Returns Expr\Func i
nstance

    // Example - $qb->expr()->some($qb2->getDql())
    public function some($subquery); // Returns Expr\Func
instance

    // Example - $qb->expr()->any($qb2->getDql())
    public function any($subquery); // Returns Expr\Func i
nstance
```

```
    // Example - $qb->expr()->not($qb->expr()->eq('u.id',
'?1'))
    public function not($restriction); // Returns Expr\Fun
c instance

    // Example - $qb->expr()->in('u.id', array(1, 2, 3))
    // Make sure that you do NOT use something similar to
$qb->expr()->in('value', array('stringvalue')) as this wil
l cause Doctrine to throw an Exception.
    // Instead, use $qb->expr()->in('value', array('?1'))
and bind your parameter to ?1 (see section above)
    public function in($x, $y); // Returns Expr\Func insta
nce

    // Example - $qb->expr()->notIn('u.id', '2')
    public function notIn($x, $y); // Returns Expr\Func in
stance

    // Example - $qb->expr()->like('u.firstname', $qb->exp
r()->literal('Gui%'))
    public function like($x, $y); // Returns Expr\Comparis
on instance

    // Example - $qb->expr()->notLike('u.firstname', $qb->
expr()->literal('Gui%'))
    public function notLike($x, $y); // Returns Expr\Compa
rison instance
```

```
    // Example - $qb->expr()->between('u.id', '1', '10')
    public function between($val, $x, $y); // Returns Expr
\Func

    /** Function objects **/

    // Example - $qb->expr()->trim('u.firstname')
    public function trim($x); // Returns Expr\Func

    // Example - $qb->expr()->concat('u.firstname', $qb->e
xpr()->concat($qb->expr()->literal(' '), 'u.lastname'))
    public function concat($x, $y); // Returns Expr\Func

    // Example - $qb->expr()->substring('u.firstname', 0,
1)
    public function substring($x, $from, $len); // Returns
 Expr\Func

    // Example - $qb->expr()->lower('u.firstname')
    public function lower($x); // Returns Expr\Func

    // Example - $qb->expr()->upper('u.firstname')
    public function upper($x); // Returns Expr\Func

    // Example - $qb->expr()->length('u.firstname')
    public function length($x); // Returns Expr\Func
```

```
    // Example - $qb->expr()->avg('u.age')
    public function avg($x); // Returns Expr\Func

    // Example - $qb->expr()->max('u.age')
    public function max($x); // Returns Expr\Func

    // Example - $qb->expr()->min('u.age')
    public function min($x); // Returns Expr\Func

    // Example - $qb->expr()->abs('u.currentBalance')
    public function abs($x); // Returns Expr\Func

    // Example - $qb->expr()->sqrt('u.currentBalance')
    public function sqrt($x); // Returns Expr\Func

    // Example - $qb->expr()->count('u.firstname')
    public function count($x); // Returns Expr\Func

    // Example - $qb->expr()->countDistinct('u.surname')
    public function countDistinct($x); // Returns Expr\Fun
c
}
```

Los repositorios

La clase Repository nos ofrece automáticamente varios métodos muy útiles para obtener registros de la base de datos:

- find
- findOneByXXX
- findByXXX
- findAll
- findOneBy
- findBy

```
$repository = $this->getDoctrine()->getRepository(Grado::class);

// Obtener un grado buscando por su primary key (normalmente "id")
$grado = $repository->find($gradoId);

// Métodos dinámicos para obtener un grado buscando por el valor de una columna
$grado = $repository->findOneById($gradoId);
$grado = $repository->findOneByNombre('Ingeniería de montes');

// Métodos dinámicos para obtener un array de objetos grado buscando por el valor de una columna
```

```php
$grados = $repository->findByNombre('Ingeniería de montes'
);

// Obtener todos los grados
$grados = $repository->findAll();

// filtrar por varios campos: una entidad
$$grado = $repository->findOneBy(
    array('nombre' => 'montes', 'credects' => 6)
);

// filtrar por varios campos: array de entidades
$$grados = $repository->findBy(
    array('nombre' => 'matematicas'),
    array('credects' => 'ASC')
);
```

Ejecutar sentencias SQL directamente

En caso de encontrarnos alguna SQL que no sepamos realizar con DQL, podemos recurrir a realizarla con SQL.

```
$em = $this->getDoctrine()->getManager();
$connection = $em->getConnection();
$statement = $connection->prepare("SELECT something FROM s
omethingelse WHERE id = :id");
$statement->bindValue('id', 123);
$statement->execute();
$results = $statement->fetchAll();
```

Evidentemente, si ejecutamos consultas de esta forma, no tendremos los datos en entidades, los tendremos en arrays de php.

Ingeniería inversa con doctrine

El primer paso para construir las clases de entidad a partir de una base de datos existente, es pedir a doctrine que inspeccione la base de datos y genere los correspondiente archivos de metadatos. Los archivos de metadatos describen las entidades que se deben generar a partir de los campos de las tablas:

> *php bin/console doctrine:mapping:import App\Entity annotation --path=src/Entity*

Las entidades generadas son clases con anotaciones y metadata, pero no tienen los métodos getters y setters.

Para generar dichos métodos, tenemos el siguiente comando:

> *bin/console make:entity --regenerate App*