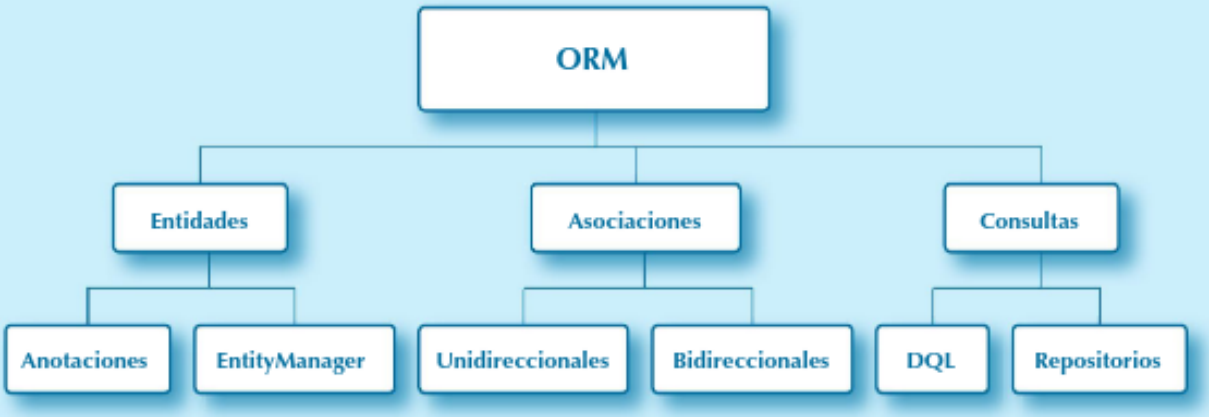
7

Mapeo objeto-relacional (ORM)

Objetivos

- ✓ Entender la utilidad de los ORM.
- ✓ Comprender las dificultades para almacenar objetos en el modelo relacional.
- ✓ Aprender a utilizar el ORM Doctrine.
- ✓ Crear entidades para representar tablas de la base de datos.
- ✓ Conocer las anotaciones más importantes en Doctrine.
- ✓ Diferenciar entre los distintos tipos de asociaciones.

Mapa conceptual



Glosario

Anotación. Metadatos que se añaden como comentarios al código fuente.

Asociación. Las entidades se relacionan mediante asociaciones. Son parecidas a las claves ajenas en el modelo relacional.

Desajuste por impedancia objeto-relacional. Con esta expresión se hace referencia a las dificultades que surgen al almacenar objetos en bases de datos relacionales.

Doctrine. ORM para PHP. Es el ORM por defecto de Symfony.

DQL. *Doctrine Query Language*. Lenguaje parecido al SQL para hacer búsquedas en Doctrine.

Entidad. Denominación que reciben las clases sincronizadas con la base de datos.

Hibernate. Uno de los ORM más extendidos para JEE. Doctrine está inspirado en él.

Mapeo objeto-relacional. El mapeo objeto-relacional consiste en asociar las clases que se manejan en la aplicación y la base de datos para simplificar el manejo de esta. También se denomina ORM al software que lo implementa.

7.1. Mapeo objeto-relacional

Mediante el mapeo objeto-relacional es posible asociar los elementos de la base de datos con los objetos de la aplicación de manera que la gestión de la base de datos sea más sencilla. La gestión no se realiza directamente sobre la base de datos, sino sobre una serie de clases que la replican.

Dentro del patrón MVC, que se verá en el próximo capítulo, los ORM se encargan del modelo. Es importante señalar que los ORM no se utilizan solo dentro del patrón MVC, también se emplean en otros tipos de aplicaciones.

En los capítulos anteriores hemos realizado el acceso a datos enviando las sentencias correspondientes como cadenas de texto que hay que construir cuidadosamente, un proceso en el que es fácil cometer errores. Además, las operaciones con las bases de datos se repiten en todas las aplicaciones cambiando el nombre de las tablas y las columnas.

Los ORM liberan al programador de muchas tareas repetitivas, generando el código necesario para comunicarse con la base de datos. Proporcionan un nivel de abstracción adicional que permite integrar diferentes orígenes de datos con un lenguaje común y facilita la reusabilidad del código. Por ejemplo, si se desarrolla una aplicación utilizando SQL Server como gestor de la base de datos, pero más adelante se decide migrar a Oracle, no habrá que adaptar el código que gestiona la base de datos a las diferencias entre fabricantes, ya que de esa parte se ocupa el ORM. Por supuesto, siempre que el ORM utilizado sea compatible con ambos sistemas.

Por ejemplo, para cargar el departamento 3 usando el ORM Doctrine, se usaría el siguiente código:

```
$dep = $entityManager->find("Departamento", 3);
```

Con la librería PDO se utilizaba:

```
$sql = 'SELECT * FROM departamentos where codDept = 3';
$usuarios = $bd->query($sql);
```

En este caso sencillo ya se aprecian las ventajas. Es el método `find()` el que se encarga de construir y ejecutar la cadena SQL correspondiente. La clase `Departamentos` contiene los métodos necesarios para realizar las operaciones de la base de datos. Siguiendo con el ejemplo anterior, es posible actualizar los departamentos modificando las propiedades del objeto y guardándolo:

```
$dep->setPresupuesto(70000);
$entityManager->flush();
```

Además de los métodos, los ORM incluyen un leguaje para hacer consultas sobre el modelo para realizar consultas complejas.

En general, cada tabla de la base de datos tiene una clase asociada, y el conjunto de estas clases constituye el modelo de la aplicación. Es posible crear las clases a partir de la base de datos utilizando asistentes y viceversa.

7.2. Doctrine

Doctrine es un ORM muy extendido para PHP. Está inspirado en Hibernate, uno de los ORM más extendidos, muy utilizado en JEE.

7.2.1. Instalación y configuración

Se puede instalar Doctrine utilizando `composer` mediante el siguiente comando:

```
composer require doctrine/orm
```

El siguiente paso consiste en crear el fichero **bootstrap.php**. Este fichero sirve para:

- Configurar la conexión con la base de datos. Los datos se introducen en el *array* `$dbparams`, en las líneas 8-13.
- Obtener un objeto de la clase `EntityManager`. Esta clase expone la interfaz pública de Doctrine. En la última línea se crea el objeto `$entityManager`, que estará disponible en los ficheros que incluyan a **bootstrap.php**.

```
<?php
require_once "../vendor/autoload.php";
use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;
$paths = array("./src");
$isDevMode = true;
// configuración de la base de datos
$dbParams = array(
'driver' => 'pdo_mysql',
'user' => 'root',
'password' => '',
'dbname' => 'doctrine',
'host' => 'localhost',
);
$config = Setup::createAnnotationMetadataConfiguration($paths,
    $isDevMode, null, null, false);
$entityManager = EntityManager::create($dbParams, $config);
```

TOMA NOTA

En la segunda línea se incluye el fichero **autoload.php**, que carga Doctrine. Hay que ajustar la ruta según dónde esté la carpeta.

7.2.2. Entidades

El elemento básico de Doctrine son las entidades. Las entidades son clases que están asociadas con la base de datos. Los atributos (al menos algunos de ellos) de estas clases son persistentes. Es decir, se almacenan en una base de datos.

La relación entre las entidades y la base de datos se describe mediante metadatos que pueden añadirse con anotaciones en la propia entidad o con un fichero externo en formato XML. En este capítulo se utiliza la primera opción.

Las anotaciones se introducen dentro de bloques de comentarios DocBlock. Aparecen precedidas por una arroba (@) y pueden tener atributos. Por ejemplo, para asociar una clase con una tabla se utilizan las anotaciones `@Entity` y `@Table`. El bloque se sitúa justo encima de la declaración de la clase.

```
/**
* @ORM\Entity
* @ORM\Table(name="tabla")
*/
class Clase
{
```

La anotación `@Entity` simplemente dice que la clase es una entidad. La anotación `@Table` sirve para indicar con qué tabla se relaciona (con el atributo `name`). Ambas anotaciones tienen otros atributos que no se usan en el ejemplo.

En general, en las anotaciones hay que especificar:

- La tabla con la que está mapeada la entidad.
- La columna con la que está mapeado cada atributo persistente.
- Los modificadores de cada atributo, por ejemplo, que se trata un valor autogenerado.
- Las asociaciones con otras entidades. Las asociaciones son la contraparte de las claves ajenas.

Las entidades son clases normales con las siguientes restricciones:

a) Sus atributos tienen que ser `protected` o `private`.
b) En general hay que hacer *getters* y *setters* para todos los atributos salvo los que se correspondan con una columna autogenerada, por ejemplo, los campos de autoincremento en MySQL. Para estos campos no se hace *setter*.

TOMA NOTA

En este capítulo se usan las siguientes tablas de la base de datos "doctrine".

Equipo(Id, Nombre, Socios, Fundacion, Ciudad)
Jugador(Id, nombre, Apellidos, Edad, Equipo)

Los atributos id son clave primaria. La columna Equipo en Jugador es clave ajena.

La clase Equipo es una entidad sincronizada con la tabla Equipo. Tiene un atributo por cada columna de la tabla. Como se puede ver, sobre cada atributo hay un bloque con la anotación `@Column` para indicar el tipo de dato.

Para el primer atributo hay además otras dos anotaciones:

- `@Id` indica que el atributo es clave primaria.
- `@GeneratedValue` indica que es un valor autogenerado.

En este caso los nombres de los atributos de la clase coinciden con los de las columnas de la tabla. Si no fuera así, se utiliza el atributo `name` dentro de la anotación `@Column` para indicar con qué columna se asocia el atributo.

@ORM\Column(type="integer", name ="columna")

```php
<?php
// src/Equipo.php
use Doctrine\ORM\EntityRepository;
use Doctrine\ORM\Mapping as ORM;;
/**
* @ORM\Entity
* @ORM\Table(name="equipo")
@ORM\Entity(repositoryClass="EquipoRepository")
*/
class Equipo{
    /** @ORM\Id @ORM\Column(type="integer") @ORM\GeneratedValue **/
    private $id;
    /** @ORM\Column(type="string") **/
    private $nombre;
    /** @ORM\Column(type="integer") **/
    private $fundacion;
    /** @ORM\Column(type="integer") **/
    private $socios;
    /** @ORM\Column(type="string") **/
    private $ciudad;
    public function getId()
    {
        return $this->id;
    }
    public function getNombre()
    {
        return $this->nombre;
    }
    public function setNombre($nombre)
    {
        $this->nombre = $nombre;
    }
    public function getFundacion()
    {
        return $this->fundacion;
    }
    public function setFundacion($fundacion)
    {
        $this->fundacion = $fundacion;
    }
    public function getSocios()
    {
        return $this->socios;
    }
    public function setSocios($socios)
    {
        $this->socios = $socios;
    }
    public function getCiudad()
    {
        return $this->ciudad;
    }
    public function setCiudad($ciudad)
    {
        $this->ciudad = $ciudad;
    }
}
```

En el ejemplo siguiente se puede ver cómo manejar las entidades. En las primeras líneas se incluyen tanto la entidad que se quiere usar como el fichero **bootstrap.php**. En ese fichero se declara `$entityManager`, que tiene una serie de métodos para, como su nombre indica, gestionar las entidades.

Por ejemplo, el método `find()` sirve para buscar una entidad por clave. Recibe el nombre de la clase y un valor. Para buscar el equipo con id 1 se usaría:

```
$eq = $entityManager->find("Equipo", 1);
```

Como la entidad equipo está asociada, mediante las anotaciones, con la tabla Equipos, `find()` buscará en esa tabla algún equipo con código uno. Si existe, devuelve un objeto de la clase Equipos en la que los atributos tendrán el valor de las columnas correspondientes.

El objeto devuelto se puede manipular a través de sus métodos, como muestra el siguiente ejemplo:

```
1  <?php
2  require_once './src/Equipo.php';
3  require_once "bootstrap.php";
4  // buscar por clave primaria
5  $eq = $entityManager->find("Equipo", 1);
6  // mostrar datos
7  echo $eq->getSocios();
8  // cambiar el número de socios
9  $eq->setSocios(70000);
10 $entityManager->flush();
11 // si el equipo no existe devuelve NULL
12 $eq = $entityManager->find("Equipo", 4);
13 if(!$eq){
14     echo "Equipo no encontrado";
15 }
```

En la línea 9 se modifica el número de socios. En la línea 10 se utiliza el método `flush()`. Este método hace que se salven en la base de datos los cambios que se hayan producido en las entidades. Es en este momento cuando realmente se modifica la base de datos. Doctrine se encarga de ejecutar la sentencia SQL correspondiente.

En la línea 12 se busca un equipo con código 4, que no existe. En este caso, el método devuelve NULL.

Actividad propuesta 7.1

Escribe un programa que reciba por la URL el código de un equipo y muestre sus datos. Si el equipo no existe, tiene que mostrar un mensaje apropiado.

7.2.3. Inserción y borrado

Para realizar una inserción:

- Se crea un nuevo objeto.

- Se indica que debe almacenarse con la base de datos con el método `persist()`.
- Se llama al método `flush()` para que realmente se produzca la inserción.

```
<?php
require_once './src/Equipo.php';
require_once "bootstrap.php";
$nuevo = new Equipo();
$nuevo->setNombre('Real Madrid');
$nuevo->setFundacion(1900);
$nuevo->setSocios(50000);
$nuevo->setCiudad('Madrid');
$entityManager->persist($nuevo);
$entityManager->flush();
echo "Equipo insertado ". $nuevo->getId(). "\n";
```

Actividad propuesta 7.2

Escribe un formulario para insertar un nuevo equipo. Si hay algún error al insertar, hay que mostrar un mensaje apropiado.

Para borrar un registro:

- Se selecciona el objeto correspondiente.
- Se elimina con el método `remove()`.
- Se llama al método `flush()` para que realmente se produzca el borrado.

El ejemplo **borrar.php** borra un equipo a partir de su id, que recibe en la URL.

```
<?php
require_once "bootstrap.php";
require_once './src/Equipo.php';
$id = $_GET['id'];
/*buscar el jugador con el id indicado*/
$equipo = $entityManager->find("Equipo", $id);
if(!$equipo){
    echo "Equipo no encontrado";
}else{
    $entityManager->remove($equipo);
    $entityManager->flush();
    echo "Equipo borrado";
}
```

7.3. Asociaciones

Las asociaciones vinculan unos objetos con otros. Son la contraparte en la base de datos de objetos de las claves ajenas de las tablas, pero con diferencias importantes.

Hay varios tipos de asociaciones, se diferencian por:

- Su cardinalidad: muchos a uno, uno a muchos, uno a uno, muchos a muchos…

- Direccionalidad: unidireccionales o bidireccionales.
- Si incluyen una referencia a la propia entidad.

En este apartado se presentan las más utilizadas, especialmente si se parte de un modelo relacional ya creado.

7.3.1. Asociaciones muchos a uno unidireccionales

Este es el caso típico para una clave ajena relacional. Por ejemplo, muchos jugadores tienen un mismo equipo. Si una tabla tiene una clave ajena, en la entidad correspondiente habrá una referencia a un objeto de la entidad correspondiente a la otra tabla.

En el caso de las tablas de jugadores y equipos, esto se traduce en que la entidad Jugador tendrá un atributo que será una referencia a un objeto de la clase Equipo.

Esta es una de las diferencias principales entre las asociaciones y las claves ajenas del modelo relacional:

- La tabla de jugadores tiene una columna Equipo, que representa el equipo del jugador. Contiene el id del equipo correspondiente, que en este caso es un entero.
- En cambio, el atributo correspondiente en la clase Jugador es un objeto de clase Equipo. Es decir, el atributo equipo en la clase no contiene el código del equipo, sino una referencia al equipo correspondiente.

La clase es bastante parecida a Equipo, pero hay una novedad. El atributo equipo está anotado con:

- `@ManyToOne`, que indica que se trata de una relación *muchos a uno*. El atributo `targetEntity` especifica la entidad asociada.
- `@JoinColumn`, para indicar qué columnas se utilizan en la unión de las tablas. El atributo `name` es el nombre en la tabla que tiene la referencia (en este caso, Jugador) y `referencedColumn` es la columna de la otra tabla (Equipo).

```php
<?php
use Doctrine\ORM\Mapping as ORM;
/**
* @ORM\Entity @ORM\Table(name="jugador")
**/
class Jugador{
    /** @ORM\Id @ORM\Column(type="integer") @ORM\GeneratedValue **/
    private $id;
    /** @ORM\Column(type="string") **/
    private $nombre;
    /** @ORM\Column(type="string") **/
    private $apellidos;
    /** @ORM\Column(type="integer") **/
    private $edad;
    /**
    * @ORM\ManyToOne(targetEntity="Equipo")
```

```
     * @ORM\JoinColumn(name="equipo",
        referencedColumnName="id")
     **/
     private $equipo;
     public function getId(){
        return $this->id;
     }
     public function getNombre(){
        return $this->nombre;
     }
     public function setNombre($nombre){
        $this->nombre = $nombre;
     }
     public function getApellidos(){
        return $this->apellidos;
     }
     public function setApellidos($apellidos){
        $this->apellidos = $apellidos;
     }
     public function getEdad(){
        return $this->edad;
     }
     public function setEdad($edad){
        $this->edad = $edad;
     }
     public function getEquipo(){
        return $this->equipo;
     }
     public function setEquipo($equipo)
     {
        $this->equipo = $equipo;
     }
}
```

En el ejemplo **jugador_equipo.php** se utiliza la asociación para acceder al equipo de un jugador. Recibe el id de un jugador a través de la URL y lo busca. Si el jugador existe, muestra su nombre y el de su equipo.

```
1  <?php
2  require_once "bootstrap.php";
3  require_once './src/Equipo.php';
4  require_once './src/Jugador.php';
5  $id = $_GET['id'];
6  /*buscar el jugador con el id indicado*/
7  $jugador = $entityManager->find("Jugador", $id);
8  if(!$jugador){
9     echo "Jugador no encontrado";
10 }else{
```

```
11      echo "Nombre del jugador: ". $jugador->getNombre()."<br>";
12      $equipo = $jugador->getEquipo();
13      echo "Nombre del equipo: ". $equipo->getNombre()."<br>";
14  }
```

En la línea 7, se obtiene el objeto Jugador a partir del parámetro de la URL.

En la línea 12, `getEquipo()` devuelve una referencia al objeto correspondiente, que es de clase Equipo.

En la línea 13, se muestra el nombre del equipo utilizando el método `getNombre()`.

RECUERDA

✓ Las dos entidades tienen un método getNombre().

7.3.2. Asociaciones muchos a uno bidireccionales

En las asociaciones bidireccionales, ambas entidades reciben referencias de la entidad asociada. Esta es otra de las diferencias importantes con el modelo relacional.

En el caso de los Equipos y Jugadores, la bidireccionalidad consistiría en añadir un atributo Jugadores a la clase Equipos. Como en un equipo hay muchos jugadores, hay que almacenar muchas referencias. En lugar de un *array* normal de PHP, se usa una clase de Doctrine, `ArrayCollection`.

Veamos cómo implementarlo con las clases `EquipoBidireccional` y `JugadorBidireccional`.

La clase `EquipoBidireccional` es como equipo, pero añade este campo. También hay que añadir un constructor para inicializar el objeto `ArrayCollection`.

```
/**
* Un equipo tiene muchos jugadores
* @ORM\OneToMany(targetEntity="JugadorBidireccional",
    mappedBy="equipo")
*/
private $jugadores;
public function __construct() {
    $this->jugadores = new ArrayCollection();
}
```

En la clase Jugador la anotación del atributo `$equipo` se modifica para indicar que es una asociación inversa. Con el atributo `inversedBy` se indica el atributo de Equipo que almacenará los jugadores.

```
/**
* @ORM\ManyToOne(targetEntity="EquipoBidireccional",
    inversedBy = "jugadores")
* @ORM\JoinColumn(name="equipo", referencedColumnName="id")
*/
    private $equipo;
```

El ejemplo **probar_bidireccional.php** recibe en la URL el código de un equipo y muestra los nombres de los jugadores. Para cargar los jugadores usa el método `getJugadores()`, así que no hay que hacer una consulta. Doctrine carga los datos a través de la información contenida en las anotaciones.

```
<?php
require_once "bootstrap.php";
require_once './src/EquipoBidireccional.php';
require_once './src/JugadorBidireccional.php';
$id = $_GET['id'];
$equipo = $entityManager->find("EquipoBidireccional", $id);
if(!$equipo){
    echo "Equipo no encontrado";
}else{
    echo "Nombre del equipo: ". $equipo->getNombre()."<br>";
    $jugadores = $equipo->getJugadores();
    echo "Lista de jugadores"."<br>";
    foreach($jugadores as $jugador){
        echo "Nombre: ". $jugador->getNombre()."<br>";
    }
}
```

7.4. Consultas básicas

En Doctrine un repositorio es una clase que contiene consultas relacionadas. Se puede obtener un repositorio básico con métodos útiles para cualquier clase con el método `getRepository()` de `EntityManager`. Recibe el nombre de una entidad y devuelve un objeto de clase `EntityRepository`.

El ejemplo **findBy.php** usa algunos:

- `findBy()`. Para utilizar criterios de búsqueda, que se introducen en un *array*.
- `findOneBy()`. Como el anterior, pero devuelve solo un resultado.
- `findAll()`. Devuelve todas las filas de la tabla asociada.

```
<?php
require_once "bootstrap.php";
require_once './src/Jugador.php';
require_once './src/Equipo.php';
/*Con findBy/findOneBy:
-Jugadores con exactamente XX años..*/
echo "Jugadores con 12 años<br>";
$jugadores = $entityManager->getRepository('Jugador')
    ->findBy(array('edad' => 12));
foreach($jugadores as $jugador){
    echo "Nombre: ". $jugador->getNombre().
    " ". $jugador->getApellidos(). "<br>";
}
```

```
//Equipos de Madrid fundados en 1900.
echo "Equipos de Madrid fundados en 1900<br>";
$equipos = $entityManager->getRepository('Equipo')
    ->findBy(array(
    'fundacion' => 1900,
    'ciudad'=>'Madrid')
    );
foreach($equipos as $equipo){
echo "Nombre: ". $equipo->getNombre()."<br>";
}
/*Equipo cuyo nombre es "Real Madrid"*/
echo "Equipos cuyo nombre es 'Real Madrid'<br>";
$equipo = $entityManager->getRepository('Equipo')
    ->findOneBy(array('nombre' => 'Real Madrid'));
echo "Nombre: ". $equipo->getNombre(). " ".
    $equipo->getFundacion(). " ".
    $equipo->getCiudad()."<br>";
```

Actividad propuesta 7.3

Escribe un programa que reciba por la URL el nombre de una ciudad y muestre los datos de los equipos de esa ciudad.

7.5. DQL

Los métodos vistos hasta ahora para realizar consultas son bastante limitados. Doctrine tiene su propio lenguaje de consultas, llamado Doctrine Query Language, DQL.

Es bastante parecido a SQL, como se puede ver en estos ejemplos. Para obtener todos los datos de todos los jugadores se usaría:

SELECT j FROM jugador j

Para seleccionar el nombre los jugadores mayores de 30 años:

SELECT j.nombre FROM jugador j WHERE j.edad > 30

Para contar cuántos jugadores tienen más de 30 años:

SELECT COUNT(j.id) as num FROM jugador j WHERE j.edad > 30

Para ordenar los resultados:

SELECT j FROM jugador j ORDER BY j.edad ASC

Es necesario utilizar un alias para todas las entidades del FROM, no como en SQL, donde es opcional.

Actividad propuesta 7.4

Escribe una consulta DQL que devuelva los nombres de los equipos con más de 10.000 socios.

Si la consulta incluye más de una tabla y estas están asociadas, se pueden unir con JOIN. Por ejemplo, para mostrar el nombre de todos los jugadores junto con el de su equipo se puede utilizar:

```
SELECT j.nombre as nombre, e.nombre as equipo FROM Jugador j JOIN j.equipo e
```

En el FROM hay dos alias:

- "j", para la entidad Jugador.
- "e", para el atributo equipo de j.

Como las entidades están asociadas, "e" representa al equipo del jugador. Es un objeto de la entidad Equipo y en las consultas se puede acceder a sus atributos usando el alias.

Actividad propuesta 7.5

Escribe una consulta DQL que devuelva los nombres de los jugadores de los equipos de Madrid.

También se puede utilizar para actualizar:

```
UPDATE jugador j SET j.edad = j.edad + 1 WHERE j.edad > 20
```

Y para borrar:

```
DELETE jugador j WHERE j.edad > 30
```

Para ejecutar una consulta desde PHP:

- Se crea con `$entityManager->createQuery("SELECT …")`.
- Se ejecuta con el método `getResult()`, que devuelve un `ArrayCollection` con los resultados.

```
/*todos los jugadores, todos los datos*/
$query= $entityManager->createQuery("SELECT j FROM Jugador j");
$jugadores = $query->getResult();
```

```
foreach($jugadores as $jugador){
    echo "Nombre: ". $jugador->getNombre()."<br>";
}
```

Si la consulta devuelve un único valor, también se puede utilizar `getSingleScalarResult()`.

```
echo "Contar los jugadores mayores de 30 años<br>";
$query_contar = $entityManager->createQuery(
    "SELECT COUNT(j.id) as num
    FROM jugador j
    WHERE j.edad > 30");
$num_jugadores = $query_contar->getSingleScalarResult();
echo $num_jugadores."<br>";
```

7.6. Repositorios propios

Es posible crear clases repositorio propias, derivadas de la `EntityRepository`. Puede ser una opción para agrupar consultas que se repitan en la aplicación.

La clase `EquipoRepository` tiene un método `getLista()` que recibe el nombre de un equipo y devuelve sus jugadores.

```
class EquipoRepository extends EntityRepository{
/* devuelve una colección con los jugadores del equipo, -1 si no encuentra
el equipo*/
    public function getLista($nombre_equipo) {
        $equipo = $entityManager->getRepository('Equipo')
                                ->findOneBy(array('nombre' =>
                                        $nombre_equipo));
        if(!$equipo){
            return -1;
        }else{
            $query = $entityManager ->createQuery(
                        "SELECT j
                        FROM jugador j JOIN j.equipo e
                        WHERE e.nombre = '$nombre_equipo'");
            return $query->getResult();
        }
    }
}
```

Para asociar esta clase con la clase jugador, se utiliza la anotación el atributo `repositoryClass` de `@Entity`.

```
/**
* @ORM\Table(name="equipo")
@ORM\Entity(repositoryClass="EquipoRepository")
*/
class Equipo{
```

Recurso web

En este capítulo se ven las anotaciones más habituales, pero hay más de 30. Puedes consultar una lista completa de anotaciones y atributos en la documentación de Doctrine.

https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/annotations-reference.html

Resumen

- Para comenzar a trabajar con Doctrine hay que obtener un objeto de la clase EntityManager asociado a la base de datos.
- Cada tabla de la base de datos se asocia con una entidad. Los atributos de la clase estarán asociados a las columnas de la tabla.
- Los metadatos se introducen mediante anotaciones en las clases o con ficheros XML externos.
- Las claves ajenas del modelo relacional se representan como asociaciones en la base de datos de objetos.
- El EntityManager permite obtener un objeto repositorio asociado a una clase con getRepository().
- Este objeto tiene métodos para recuperar información de la base de datos sin tener que escribir la consulta.
- El lenguaje DQL sirve para hacer consultas más complejas.
- En general, cuando se devuelve una sola fila, se devuelve un objeto de la entidad correspondiente.
- Las consultas devuelven un objeto ArrayCollection.
- La comunicación con la base de datos se produce cuando se ejecuta flush().

Ejercicios propuestos

1. Escribe un formulario para borrar equipos usando Doctrine.
2. Escribe una consulta que devuelva todos los equipos ordenados por año de fundación.
3. Añade esta consulta a un repositorio y escribe un programa para probarlo.
4. La base de datos Doctrine también incluye la tabla:
   - Partido(Id, Local, Visitante, Goles_local, Goles_visitante, Fecha)
     Id es la clave primaria y Local y Visitante son claves ajenas de Equipo.

a) Escribe una entidad para la tabla partido incluyendo las asociaciones.
b) Escribe un programa para probarla.

5. A partir del ejercicio anterior, modifica la entidad Equipo para que la relación con Partido sea bidireccional. Escribe un programa para probarla.

6. Escribe una consulta que devuelva todos los partidos en los que el equipo "Barcelona" jugó como visitante.

Actividades de autoevaluación

1. ¿Cuál de estas afirmaciones sobre los ORM no es cierta?
   - ☐ *a)* Asocian los objetos de la aplicación con la base de datos.
   - ☐ *b)* No se basan en el modelo relacional.
   - ☐ *c)* Liberan al programador de tareas repetitivas.

2. En general, al utilizar un ORM habrá una entidad por cada:
   - ☐ *a)* Base de datos.
   - ☐ *b)* Tabla de la base de datos.
   - ☐ *c)* Usuario de la base de datos.

3. Una entidad es una clase:
   - ☐ *a)* Normal.
   - ☐ *b)* Que hereda de la clase Entity.
   - ☐ *c)* Sincronizada con la base de datos.

4. Las anotaciones son:
   - ☐ *a)* Comentarios para explicar el programa.
   - ☐ *b)* Metadatos que se incluyen en bloques de comentarios.
   - ☐ *c)* Un tipo de entidad definido por Doctrine.

5. El DQL se utiliza para:
   - ☐ *a)* Hacer consultas sobre la base de datos de objetos.
   - ☐ *b)* Definir las entidades.
   - ☐ *c)* Controlar el acceso a la base de datos.

6. En Doctrine los repositorios agrupan:
   - ☐ *a)* Entidades relacionadas.
   - ☐ *b)* Consultas relacionadas.
   - ☐ *c)* Anotaciones relacionadas.

7. Para las asociaciones se utiliza la anotación:
   - ☐ *a)* @Asociation.
   - ☐ *b)* @ID.
   - ☐ *c)* @JoinColumn.

8. El método flush() se emplea para:
   - ☐ *a)* Deshacer los cambios realizados.
   - ☐ *b)* Salvar los cambios realizados.
   - ☐ *c)* Resetear la conexión con la base de datos.

9. Los datos de conexión con la base de datos se introducen:
   - ☐ *a)* Como argumento del constructor de EntityManager.
   - ☐ *b)* En fichero bootstrap.php.
   - ☐ *c)* En las anotaciones de las entidades.

10. Para borrar una entidad:
   - ☐ *a)* Se utiliza remove().
   - ☐ *b)* Se utiliza erase().
   - ☐ *c)* Hay que utilizar una consulta DQL.

SOLUCIONES:

1. a **b** c	**5.** **a** b c	
2. a **b** c	**6.** a **b** c	**9.** a **b** c
3. a b **c**	**7.** **a** b c	**10.** **a** b c
4. a **b** c	**8.** a **b** c	