M7 - Apuntes UF3

Índice

PDO

IMPORTANTE: mencionar que son mis ejemplos, de mi práctica, se puede hacer de otras maneras.

Constructor

Ejemplo de cómo instanciar PDO. En mi caso creo un atributo $db en cada clase modelo, y creo una conexión en el constructor.

```
class IdiomaModel {
   public $db;

   public function __construct(){
        try {
            $this->db = new PDO("mysql:host=localhost;dbname=myweb",
                "usr_generic", "2024@Thos");
            $this->db->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
        }catch (PDOException $e) {
            echo $e->getMessage();
        }
   }
```

Acciones CRUD

Create

Se realiza con prepare (utilizando el bindParam) para crear.

```
public function create($idioma){
        $valors = array();
           $stmt = $this->db->prepare("INSERT INTO `tbl_idiomes` VALUES(:id,
:iso, :imatge, :actiu, :created_at, :updated_at)");
            foreach (array("id", "iso", "imatge", "actiu", "created_at",
"updated_at") as $v){
            $get = "get" . ucfirst($v);
            $valors[$v] = $idioma->$get();
            $stmt->bindParam(":".$v, $valors[$v]);
        }
        $stmt->execute();
        $lastId = $this->db->lastInsertId();
        return $lastId;
   }

public function delete($idioma){
           $stmt = $this->db->prepare("DELETE FROM `tbl_idiomes` WHERE id =
:id");
        $id = $idioma->getId();
        $stmt->bindParam(":id", $id);
        $stmt->execute();
   }
```

Read

Se utiliza query. Este método se utiliza cuando no necesitas introducir parámetros, es decir, cuando quieres obtener todos los datos, que haces un SELECT * FROM… Si quieres hacer un getById o lo que sea, tienes que utilizar el prepare, y asignar el parámetro, como en el create. En este caso se podría mejorar, para no tener que hacer un array e irlos convirtiendo en objetos. Se podría emplear otro método para que devolvieran objetos (se explica más adelante).

```
public function read($idioma = null){
      $idiomes = array();
      $res = $this->db->query("SELECT * FROM `tbl_idiomes` ORDER BY `id`
DESC");
       while ($col = $res->fetch(PDO::FETCH_ASSOC)) {
           $i = new Idioma($col['id'], $col['iso'], $col['imatge'],
$col['actiu'], $col['created_at'], $col['updated_at']);
           $idiomes[] = $i;
       }

       return $idiomes;
   }
```

Update

Básicamente es lo mismo que el create. Importante mencionar que el $get lo estoy haciendo porque los atributos del objeto de negocio Idioma, son privados. En caso de que sean públicos te puedes saltar esto y solo hacer: $stmt->bindParam(':'.$v,$$v);

```
public function update($idioma){
      $valors = array();
      $stmt = $this->db->prepare("UPDATE `tbl_idiomes` SET iso = :iso,
imatge = :imatge, actiu = :actiu WHERE id = :id");
       foreach (array("iso", "imatge", "actiu", "id") as $v){
           $get = "get" . ucfirst($v);
           $valors[$v] = $idioma->$get();
           $stmt->bindParam(":".$v, $valors[$v]);
       }
       $stmt->execute();
   }
```

Delete

```
public function delete($idioma){
   $stmt = $this->db->prepare("DELETE FROM `tbl_idiomes` WHERE id = :id");
   $id = $idioma->getId();
   $stmt->bindParam(":id", $id);
   $stmt->execute();
}
```

Apuntes rápidos

IMPORTANTE: Hay cosas importantes, o que pueden ser muy útiles, otras no.
Existen dos métodos para obtener los errores SQL que se generen:

- errorCode() → devuelve el código de error.
- errorInfo() → devuelve el código de error + el porqué.

Se puede poner para que la base de datos tire excepciones, porque PDO no deja de estar Orientado a Objetos. Para ello:

```
$db->setAttribute(
   PDO::ATTR_ERRMODE,
   PDO::ERRMODE_EXCEPTION
);
```

Los resultados que se obtienen con las query, pueden venir de diferentes maneras:

- Array (Numérico o Asociativo).
- String (Conjuntos o una sola columna).
- Objetos (stdClass, objetos de una clase, o dentro de un objeto existente).
- Funciones de Callback.
- Lazy fetching.
- Iteradores.

Ejemplo de Array:

```
$res = $db->query( "SELECT * FROM tbl_comptes" );
$res = $db->query( "SELECT * FROM foo" );
while ($row = $res->fetch( PDO::FETCH_NUM)){
    // $row == array with numeric keys
}
$res = $db->query( "SELECT * FROM foo" );
while ($row = $res->fetch( PDO::FETCH_ASSOC)){
    // $row == array with associated (string) keys
}
$res = $db->query( "SELECT * FROM foo" );
while ($row = $res->fetch( PDO::FETCH_BOTH)){
    // $row==array with associated & numeric keys
}
```

Ejemplo de String:

```
$u = $db->query("SELECT users WHERE
login='login' AND password='password'");

//fetch(PDO::FETCH_COLUMN)
if ($u->fetchColumn()) { // retorna un string
    // login OK
} else {
    // authentication failure
}
```

Ejemplo de Object:

```
// Obtienes una instancia donde el nombreDeColumna ==
nombreDeLaPropiedad
```

```php
$res = $db->query("SELECT * FROM foo");
while ($obj = $res->fetch(PDO::FETCH_OBJ)) {
// $obj == instance of stdClass
}

// Obtienes una fila como una instancia de una clase preexistente.
$u = new userObject();
$res = $db->query("SELECT * FROM users");
$res->setFetchMode(PDO::FETCH_INTO, $u);
while ($res->fetch()) {
   // will re-populate $u with row values
}
```

Ejemplo de Class:

```php
$res = $db->query( "SELECT * FROM foo" );
$res->setFetchMode( PDO::FETCH_CLASS,"className",
array("optional"=>"Constructor Params" )
);
while ($obj = $res->fetch()) {
    // $obj == instance of className
}
```

Ejemplo de Iterator:

```php
// PDOStatement implementa una interfaz Iterator, que permite recorrer
el resultado sin necesidad de hacer servir un método.
$res = $db->query(
"SELECT * FROM users",
PDO::FETCH_ASSOC
);
foreach ($res as $row) {
   // $row == associated array representing
   // the row's values.
}
```

Ejemplo de CallBack:

```php
// PDO tiene un método donde cada resultado se procesa como una función.
function draw_message($subject,$email) { //...
}
$res = $db->query("SELECT * FROM msg");
$res->fetchAll(
PDO::FETCH_FUNC,
   "draw_message"
);
```

Para intervenir la inyección de SQL, se puede usar un método llamado quote() o mucho más familiar, que serían los Prepared Statement. *Yo recomiendo Prepared Statement*. Ejemplo de quote():

```
$qry = "SELECT * FROM users WHERE login="
.$db->quote($_POST['login']) ."AND passwd="
.$db->quote($_POST['pass']);
```

Para obtener datos de manera parcial, se emplea un método llamado closeCursor():

```
$res = $db->query("SELECT * FROM users");
foreach ($res as $v) {
    if ($res["name"] == "end") {
        $res->closeCursor();
        break;
    }
}
```

También se pueden hacer transacciones, como en SQL:

```
$db->beginTransaction();
    if ($db->exec($qry) === FALSE) {
       $db->rollback();
    }
$db->commit();
```

También se puede usar metadata para obtener los datos, pero dudo que lo lleguemos a usar. Igualmente pongo los ejemplos:

```
$res = $db->query($qry);
$ncols = $res->columnCount();
for ($i=0; $i < $ncols; $i++) {
    $meta_data = $stmt->getColumnMeta($i);
}
```

Existen varios métodos de los cuales se puede extraer información de la metadata:

- native_type → Devuelve el tipo de dato.
- driver:decl_type → Devuelve el tipo de dato que tiene la tabla de la base de datos.
- flags → Devuelve los flags de la columna en forma de array.
- name → Devuelve el nombre de la columna de la base de datos sin normalización.
- len → devuelve el tamaño máximo que permite la columna. Si no tiene es -1.
- precision → El número que tiene la columna.
- pdo_type → El tipo de la columna según PDO como uno de los PDO_PARAM constants.

Por último, existen dos métodos. Uno de ellos es muy útil:

- lastInsertId() → devuelve el id del último elemento insertado en la base de datos.
- getAttribute() → Puedes obtener datos sobre la base de datos.

```php
$db->getAttribute( PDO::ATTR_SERVER_VERSION);
// Database Server Version
$db->getAttribute( PDO::ATTR_CLIENT_VERSION);
// Client Library Server Version
$db->getAttribute( PDO::ATTR_SERVER_INFO);
// Misc Server information
$db->getAttribute( PDO::ATTR_CONNECTION_STATUS);
// Connection Status
```

Doctrine 2

Doctrine actúa como modelo. Lo que hemos estado realizando en las prácticas:

Instalación

La instalación de doctrine se realiza de la siguiente manera:

1. Crear un proyecto PHP nuevo
2. Este punto se puede realizar de dos maneras:
    a. Composer init
    Si lo quieres hacer desde 0, es decir, crear el composer.json con consola, debes acceder al directorio del proyecto con la terminal.

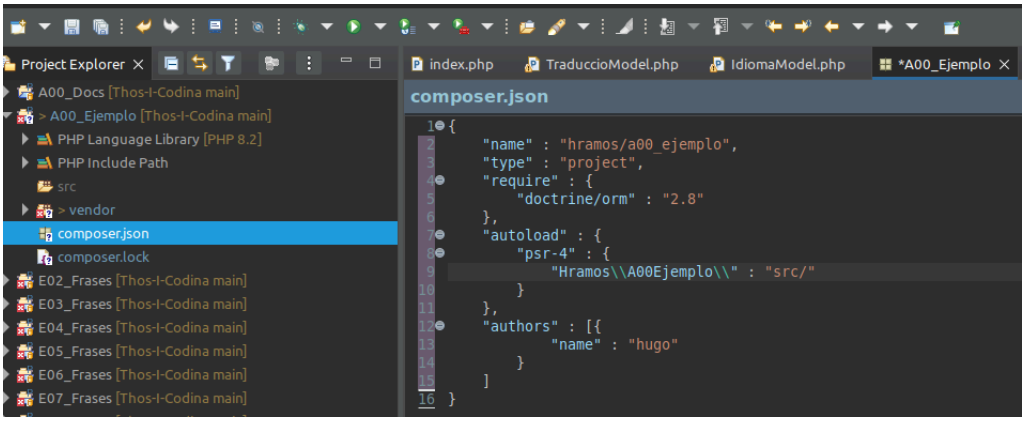
```
hramos@hramos-PC:~$ cd Thos-I-Codina/DAW-2/M7/eclipse-workspace/A00_Ejemplo/
hramos@hramos-PC:~/Thos-I-Codina/DAW-2/M7/eclipse-workspace/A00_Ejemplo$ composer init

  Welcome to the Composer config generator


This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [hramos/a00_ejemplo]:
Description []:
Author [Hugo Ramos Montesinos <hugodarken7@gmail.com>, n to skip]: hugo
Minimum Stability []:
Package Type (e.g. library, project, metapackage, composer-plugin) []: project
License []:

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]? y
Search for a package: orm
```

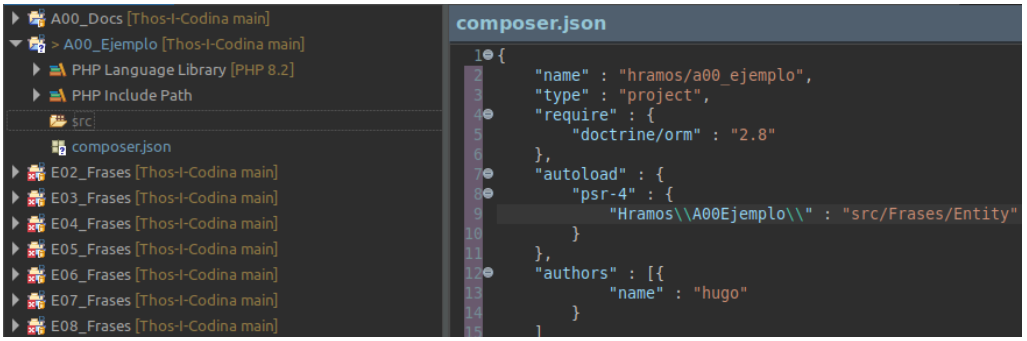
```
Found 15 packages matching orm

   [0] doctrine/orm
   [1] gedmo/doctrine-extensions
   [2] doctrine/persistence
   [3] doctrine/doctrine-bundle
   [4] symfony/orm-pack
   [5] nelmio/alice
   [6] illuminate/database
   [7] cakephp/cakephp
   [8] beberlei/doctrineextensions
   [9] sonata-project/doctrine-orm-admin-bundle
  [10] laravel-doctrine/orm
  [11] topthink/think
  [12] topthink/framework
  [13] theofidry/alice-data-fixtures
  [14] scienta/doctrine-json-functions

Enter package # to add, or the complete package name if it is not listed: 0
Enter the version constraint to require (or leave blank to use the latest version): 2.8
Search for a package:
Would you like to define your dev dependencies (require-dev) interactively [yes]? n
```

```
Add PSR-4 autoload mapping? Maps namespace "Hramos\A00Ejemplo" to the entered relative path. [src/, n to
skip]: src/

{
    "name": "hramos/a00_ejemplo",
    "type": "project",
    "require": {
        "doctrine/orm": "2.8"
    },
    "autoload": {
        "psr-4": {
            "Hramos\\A00Ejemplo\\": "src/"
        }
    },
    "authors": [
        {
            "name": "hugo"
        }
    ]
}

Do you confirm generation [yes]? y
Would you like to install dependencies now [yes]? y
```

Si todo va bien, en principio lo tendrías instalado. Hecho esto, si te piden que el proyecto tenga un namespace, debes hacer lo siguiente:

- Ir al proyecto, y refrescarlo
- Ir al composer-json
- Modificar el autoload, dentro de psr-4
- Cambiar /src, por el namespace que quieras, guardar el composer.json, y borrar la carpeta vendor y composer.lock





- Luego ir al directorio y hacer un composer install en la consola

```
hramos@hramos-PC:~/Thos-I-Codina/DAW-2/M7/eclipse-workspace/A00_Ejemplo$ composer install
No composer.lock file present. Updating dependencies to latest instead of installing from
https://getcomposer.org/install for more information.
```

Esta sería la primera manera. La segunda es más sencilla:

b. Copiar un composer.json de otro proyecto o escribirlo a mano. Luego ir a la terminal, al directorio raíz del proyecto y hacer un composer install (no pongo captura porque ya está arriba).

3. Una vez hecho esto, copiar la carpeta **config** de los otros proyectos y el fichero de **bootstrap** en el src. IMPORTANTE, dentro de la carpeta config, está el fichero **config.php,** dentro tiene escrito los parámetros de la base de datos, es importante que pongas los tuyos. En principio, al estar hecho de las otras prácticas, no haría falta modificarlo.

Actualización

Se puede actualizar de dos maneras:

1. Cambias el composer.json, borras la carpeta vendor y el composer.lock, para posteriormente poner composer install.
2. La otra forma es hacer un composer update (esto no siempre funciona).

Entidades

- No se puede implementar __clone() o __wakeup().
- No se puede usar func_get_args()
- Los atributos deben ser privados, y deben tener getters y setters.
- No pueden ser final, ni contener métodos final.
- Si creas una instancia con **new** no está manejada por el EntityManager, si la haces con **managed** si.
- Se pueden manejar de diferentes maneras:
  - Anotaciones → Esta es la que usamos nosotros.
  - XML
  - Yaml
  - PHP Metadata

Ejemplo de una entidad con anotaciones (sin relaciones)

```
/**
* Frase
*
* @Entity
* @Table(name="tbl_frases")
```

```
*
*/
class Frase {
    /**
     * @var int
     *
     * @Id
     * @GeneratedValue(strategy="AUTO")
     * @Column(type="integer", nullable=false)
     */
    private $id;

    /**
     * @var Autor
     *
     * @ManyToOne(targetEntity="Autor", inversedBy="frases")
     */
    private $autor;

    /**
     * @var string
     *
     * @column(type="string", length=250, nullable=false)
     */

    public function getId(){
        return $this->id;
    }
    public function getAutor(){
        return $this->autor;
    }
    public function getTexto(){
        return $this->texto;
    }
    public function setId($id){
        $this->id = $id;
    }
    public function setAutor($autor){
        $this->autor = $autor;
    }
    public function setTexto($texto){
        $this->texto = $texto;
    }
```

Anotaciones

Existen diferentes tipos de anotaciones, y cada una se utiliza de una manera. En principio estas son todas las anotaciones existentes. IMPORTANTE: Los atributos adjuntos en cada una de las anotaciones, pueden ir juntos, mientras estén separados por comas.

- Recomendable que los atributos apuntados vayan juntos.
- Posible que **NO ENTREN** en el examen.
- Posible que **ENTREN** en el examen.

Anotación	Descripción
@Entity	Define que es una entidad que debe persistir en la base de datos. Hay varios ejemplos: - @Entity - @Entity(repositoryClass=”Frases\Repository\EntityRepository”)
@Table	Define la tabla de la base de datos asociada a la entidad. - @Table(name=”nombre”) → Especifica el nombre de la tabla. - @Table(schema=”public”) → Esquema de la tabla. - @Table(name="autores", indexes={@Index(name="nombre_dni_idx", columns={"nombre", "dni"})}) → Indices para la tabla. - @Table(name="autores", uniqueConstraints={@UniqueConstraint(name="dni_unico", columns={"dni"})}) → Restricciones de la tabla. - @Table(name="autores", options={"comment"="Tabla de autores"}) → Añade comentarios. **Estas no creo que las pida:** - @Table(name="autores", inheritanceType="JOINED") → Tipo de herencia de la tabla. - @Table(name="autores",discriminatorMap=@DiscriminatorMap({"autor"="Autor", "editor"="Editor"})) → Discrimina una tabla (?) - @Table(name="autores", catalog="biblioteca") → Catálogo de una tabla. - Table(name="autores", primaryKey={@PrimaryKey(name="id")}) → Define una clave primaria en la tabla.
@Column	Define una columna en la tabla - @Column(type=”string”) → El tipo que tendrá la columna. Existen varios tipos: intenger, string, datetime, text, boolean, float, decimal, array… - @Column(length=255) → La longitud de la columna. - @Column(unique=true) → Si la columna es única. Por defecto no lo es. - @Column(nullable=true) → Si puede ser nula, si es false, no. - @Column(options={"comment"="Nombre del autor"}) → Opciones adicionales. - @Column(name=”nombre”) → El nombre que tendrá, si no se pone esta propiedad, la columna adquiere el nombre que tiene el atributo. **Estas no creo que las pida:** - @Column(precision=10) → Precisión de la columna. - @Column(scale=2) → Escala de la columna. - @Column(columnDefinition="VARCHAR(255) NOT NULL") → Define la columna como si fuera SQL.

@Id	Define la columna como una clave primaria.
@GeneratedValue	Define la estrategia de generación de la clave primaria. - @GeneratedValue(strategy=”AUTO”) → Esta es la común, define que utilice la estrategia propia de la base de datos, en el caso de MySQL es sumar +1 al id. Existen otras como SEQUENCE, IDENTITY, NONE, UUID, CUSTOM… **Estas no creo que las pida:** - @GeneratedValue(strategy="SEQUENCE", generator=”secuela_id”) → Especifica el nombre del generador de secuencias que se utilizará. - @GeneratedValue(strategy="SEQUENCE", generator=”secuela_id”, allocationSize=1) → Esta opción se utiliza cuando la estrategia de generación es "SEQUENCE". Especifica el tamaño de la asignación para el generador de secuencias.
@Version	Define una columna de versión para el control de concurrencia optimista.
@InheritanceType	Define el tipo de herencia de la entidad. - @InheritanceType("JOINED") → Tipo de entidad. Existen varios: SINGLE_TABLE, JOINED, TABLE_PER_CLASS…
@DiscriminatorColumn	Define la columna discriminadora para la herencia de entidad. - @DiscriminatorColumn(name="tipo") → Nombre de la columna discriminatoria. - @DiscriminatorColumn(type="string") → El tipo que tiene la columna discriminatoria. Imagino que es igual que cuando pones el tipo de la columna.
@DiscriminatorMap	Define el mapa discriminador para la herencia de entidad. - @DiscriminatorMap({"autor"="Autor", "editor"="Editor"})
@MappedSuperclass	Define que la clase es una superclase mapeada.
@Embeddable	Define que la clase es incrustable.
@Embedded	Define una propiedad como incrustada.
@UniqueConstraint	Define una restricción única para una tabla. - @UniqueConstraint(name="dni_unico") → Nombre de la restricción única. - @UniqueConstraint(columns={"dni"}) → Columna de la restricción única.
@Index	Define un índice para una tabla. - @Index(name="nombre_dni_idx") → Nombre del índice. - @Index(columns={"nombre", "dni"}) → Columna del índice.
@AttributeOverrides / @AttributeOverride	Permiten anular los detalles de mapeo de una propiedad incrustada.

ALERTA: Algunas de estas anotaciones y opciones solo son aplicables en ciertos contextos. Por ejemplo, mappedBy solo se puede usar en el lado no propietario de una relación bidireccional. @JoinColumn y @JoinTable solo se pueden usar en el lado propietario de una relación. @GeneratedValue solo se puede usar en una propiedad que también esté anotada con @Id.

Relaciones

En Doctrine existen diferentes relaciones entre tablas. Se podría decir, que es la cardinalidad que existe entre tablas, aunque no estoy muy seguro de esto, solo que yo lo entiendo así. Estas relaciones se establecen en la clase Entidad. En nuestro caso, como estamos utilizando anotaciones, las relaciones se establecen con estas. Las relaciones que hay en Doctrine pueden ser:

- OneToOne
- OneToMany
- ManyToOne
- ManyToMany

Antes de empezar a explicar cada relación de Doctrine, es necesario saber la diferencia entre una relación **UNIDIRECCIONAL** y una relación **BIDIRECCIONAL**:

UNIDIRECCIONAL: Es aquella relación que solo una entidad sabe de la existencia de la otra. Esto extrapolado a código para que sea más sencillo de entender, solo una de las dos entidades tiene el atributo que las relaciona. Como ejemplo se puede poner el siguiente:

Supongamos que tienes dos entidades: Producto y Categoría. Cada producto pertenece a una categoría, pero una categoría no necesita saber qué productos contiene. Esto sería una relación unidireccional.

```
// Entidad Producto
class Producto
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Categoria")
     * @ORM\JoinColumn(nullable=false)
     */
    private $categoria;

    // ...
}

// Entidad Categoría
class Categoria
{
    // ...
}
```

En este caso, un producto puede acceder a su categoría a través de la propiedad $categoria, pero una categoría no tiene una referencia directa a los productos que contiene. Esto hace que la relación sea unidireccional.

BIDIRECCIONAL: Es aquella relación donde ambas entidades saben de la existencia de la otra. Extrapolado a código, sería que ambas entidades tienen un atributo que les relaciona con la otra. Un ejemplo de esto podría ser:

Supongamos que tienes dos entidades en tu aplicación: Usuario y Publicación. Cada usuario puede tener varias publicaciones y cada publicación pertenece a un único usuario. Esto sería una relación bidireccional porque tanto el usuario como la publicación tienen conocimiento de la relación entre ellos.

```
// Entidad Usuario
class Usuario
{
    // ...

    /**
     * @ORM\OneToMany(targetEntity="App\Entity\Publicacion",
mappedBy="usuario")
     */
    private $publicaciones;

    // ...
}

// Entidad Publicación
class Publicacion
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Usuario",
inversedBy="publicaciones")
     * @ORM\JoinColumn(nullable=false)
     */
    private $usuario;

    // ...
}
```

En este ejemplo, un usuario puede acceder a sus publicaciones a través de la propiedad $publicaciones, y una publicación puede acceder al usuario que la creó a través de la propiedad $usuario.

OneToOne

Cada Entidad principal solo puede tener una asociada. Ejemplo: Un usuario solo puede tener un perfil.

Ejemplo (OneToOne unidireccional)

```
/**
* @ Entity
*/
class User
{
    /**
     * @Id
     * @GeneratedValue
     * @Column(type="integer")
     */
    protected $id;

    /**
     * @OneToOne(targetEntity="Profile", cascade={"persist", "remove"})
     */
    private $profile;

    // aquí deberían incluirse los getters y setters...
}



/**
* @ Entity
*/
class Profile
{
    /**
     * @Id
     * @GeneratedValue
     * @Column(type="integer")
     */
    protected $id;

    // aquí deberían incluirse los getters y setters...
}
```

Anotaciones

Anotación	Descripción
@OneToOne	Define una relación uno a uno. - @OneToOne(targetEntity="Perfil") → Clase de la entidad objetivo, es decir, a la que apuntas. - @OneToOne(mappedBy="usuario") → El atributo que tiene la otra entidad que referencia a la tuya. - @OneToOne(targetEntity="Perfil", cascade={"persist", "remove"}) → Define las operaciones en cascada, como en SQL. Existen varios parámetros que se pueden asignar: persist, remove, merge, detach, all. - @OneToOne(targetEntity="Perfil", cascade={"all"}) → Otra forma de hacer lo anterior. - @OneToOne(targetEntity="Perfil", orphanRemoval=true) → Define si las entidades huérfanas se deben eliminar. - @OneToOne(targetEntity="Usuario", inversedBy="perfil") → Apunta al atributo que tiene la clase propietaria, es decir, como en el ejemplo anterior se establece que usuario es la clase padre y hace el mappedBy perfil referenciando al atributo usuario de perfil, aqui se hace la inversa, dónde perfil apunta al atributo perfil de usuario. (owner) **Esto no creo que lo pida:** - @OneToOne(targetEntity="Perfil", fetch="EAGER") → Define cómo se obtiene la entidad relacionada. Existen varios parámetros: EAGER, LAZY.

Reglas de Unidireccional y Bidireccional

OneToOne puede ser unidireccional, es decir, solo una entidad conoce de la existencia de la otra, pero no viceversa, y también puede ser bidireccional, es decir que ambas entidades saben de la existencia de la otra.

OneToMany

Una entidad principal puede tener varias entidades asociadas. Ejemplo: Un autor puede tener varias frases.

Ejemplo (OneToMany bidireccional)

Aquí lo importante, es que el autor puede tener un array de frases, por lo que se debe crear un método que sea capaz de añadir las frases a su array de frases. Este array se debe inicializar como una ArrayCollection en el constructor.

```
/**
* Autor
*
* @Entity
* @Table(name="tbl_autors")
*/
class Autor {
   /**
    * @var int
```

```
     *
     * @Id
     * @GeneratedValue(strategy="AUTO")
     * @Column(type="integer", nullable=false)
     */
    private $id;

    /**
     * @var string
     *
     * @column(type="string", length=50, nullable=false)
     */
    private $nombre;

    /**
     * @var string
     *
     * @column(type="string", length=90)
     */
    private $descripcion;

    /**
     * @var Frase[]
     *
     * @OneToMany(targetEntity="Frase", mappedBy="autor")
     */
    private $frases;

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

    /**
     * Add frase
     *
     * @param Frase $frase
     * @return Autor
     */
    public function addFrase(Frase $frase){
        $this->frases[] = $frase;
        $frase->setAutor($this);
        return $this;
    }
```

Frase por el otro lado, solo tiene un autor, por lo que no es necesario el array.

```
/**
* Frase
*
* @Entity
* @Table(name="tbl_frases")
*
*/
```

```php
class Frase {
    /**
     * @var int
     *
     * @Id
     * @GeneratedValue(strategy="AUTO")
     * @Column(type="integer", nullable=false)
     */
    private $id;

    /**
     * @var Autor
     *
     * @ManyToOne(targetEntity="Autor", inversedBy="frases")
     */
    private $autor;
```

Anotaciones

Anotación	Descripción
@OneToMany	Define una relación uno a muchos. - @OneToMany(targetEntity="Frase") → Clase de la entidad objetivo, es decir, a la que apuntas. - @OneToMany(mappedBy="autor") → El atributo que tiene la otra entidad que referencia a la tuya. - @OneToMany(targetEntity="Frase", cascade={"persist", "remove"}) → Define las operaciones en cascada, como en SQL. Existen varios parámetros que se pueden asignar: persist, remove, merge, detach, all. - @OneToMany(targetEntity="Frase", cascade={"all"}) → Otra forma de hacer lo anterior. - @OneToMany(targetEntity="Frase", orphanRemoval=true) → Define si las entidades huérfanas se deben eliminar. - @OneToMany(targetEntity="Autor", inversedBy="frases") → Apunta al atributo que tiene la clase propietaria. (owner)

Reglas de Unidireccional y Bidireccional

Normalmente es bidireccional. Aunque técnicamente puedes definir una relación OneToMany unidireccional. Doctrine no la soporta directamente y tendrías que usar una tabla de unión, lo que en realidad la convierte en una relación ManyToMany.

ManyToOne

Varias entidades pueden estar asociadas a una entidad principal. Ejemplo: Varias frases pueden estar asociadas a un autor.

Ejemplo (ManyToOne bidireccional)

El ejemplo es el mismo que el OneToMany, pero aquí hay que fijarse en el código de Frase, concretamente en la variable de autor.

Anotaciones

Anotación	Descripción
@ManyToOne	Define una relación uno a muchos. - @ManyToOne(targetEntity="Autor") → Clase de la entidad objetivo, es decir, a la que apuntas. - @ManyToOne(mappedBy="frases") → El atributo que tiene la otra entidad que referencia a la tuya. - @ManyToOne(targetEntity="Autor", cascade={"persist", "remove"}) → Define las operaciones en cascada, como en SQL. Existen varios parámetros que se pueden asignar: persist, remove, merge, detach, all. - @ManyToOne(targetEntity="Autor", cascade={"all"}) → Otra forma de hacer lo anterior. - @ManyToOne(targetEntity="Autor", orphanRemoval=true) → Define si las entidades huérfanas se deben eliminar. - @ManyToOne(targetEntity="Frase", inversedBy="autor") → Apunta al atributo que tiene la clase propietaria. (owner)

Reglas de Unidireccional y Bidireccional

Puede ser unidireccional (el lado "muchos" conoce al lado "uno") o bidireccional (ambos lados se conocen entre sí).

IMPORTANTE: En resumen, ManyToOne y OneToMany se utilizan juntos para establecer una relación bidireccional entre dos tipos de entidades, pero el lado ManyToOne es el propietario de la relación y es el que tiene la clave foránea en la base de datos.

ManyToMany

Varias entidades pueden estar asociadas a varias entidades. Ejemplo: Varias frases pueden tener varios temas.

Ejemplo (ManyToMany bidireccional)

```php
namespace Frases\Entity;
use Doctrine\Common\Collections\ArrayCollection;
/**
* Frase
*
* @Entity
* @Table(name="tbl_frases")
*
*/
class Frase {
   /**
    * @var int
    *
    * @Id
    * @GeneratedValue(strategy="AUTO")
    * @Column(type="integer", nullable=false)
    */
   private $id;

   /**
```

```
     * @var Autor
     *
     * @ManyToOne(targetEntity="Autor", inversedBy="frases")
     */
    private $autor;

    /**
     * @var string
     *
     * @column(type="string", length=250, nullable=false)
     */
    private $texto;

    /**
     * @var Tema[]
     *
     * @ManyToMany(targetEntity="Tema")
     * @JoinTable(name="tbl_frases_temes")
     */
    private $temes;

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

    /**
     * Add Tema
     *
     * @param Tema $tema
     * @return Frase
     */
    public function addTema(Tema $tema){
        $this->temes[] = $tema;
        return $this;
    }

    public function setTemes($temes){
        $this->temes = $temes;
    }
}

namespace Frases\Entity;
use Doctrine\Common\Collections\ArrayCollection;
/**
* Tema
*
* @Entity
* @Table(name="tbl_temes")
*/
class Tema {
    /**
     * @var int
     *
```

```
     * @Id
     * @GeneratedValue(strategy="AUTO")
     * @Column(type="integer", nullable=false)
     */
    private $id;

    /**
     * @var string
     *
     * @column(type="string", length=50, nullable=false)
     */
    private $nombre;

    /**
     * @var Frase[]
     *
     * @ManyToMany(targetEntity="Frase", mappedBy="temes")
     * @JoinTable(name="tbl_frases_temes")
     */
    private $frases;

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

    /**
     * Add Frase
     *
     * @param Frase $frase
     * @return Tema
     */
    public function addFrase(Frase $frase){
        $this->frases[] = $frase;
        return $this;
    }

    public function setFrases($frases){
        $this->frases = $frases;
    }
}
```

Anotaciones

Anotación	Descripción
@ManyToOne	Define una relación muchos a muchos. - @ManyToMany(targetEntity="Frase") → Clase de la entidad objetivo, es decir, a la que apuntas. - @ManyToMany(mappedBy="temes") → El atributo que tiene la otra entidad que referencia a la tuya. - @ManyToMany(targetEntity="Frase", cascade={"persist", "remove"}) → Define las operaciones en cascada, como en SQL. Existen varios parámetros que se pueden asignar: persist,

	remove, merge, detach, all. - @ManyToMany(targetEntity="Frase", cascade={"all"}) → Otra forma de hacer lo anterior. - @ManyToMany(targetEntity="Frase", orphanRemoval=true) → Define si las entidades huérfanas se deben eliminar. - @ManyToMany(targetEntity="Tema", inversedBy="frases") → Apunta al atributo que tiene la clase propietaria. (owner)

Reglas de Unidireccional y Bidireccional

Puede ser unidireccional (una entidad conoce a la otra) o bidireccional (ambas entidades se conocen entre sí).

IMPORTANTE: En resumen, todas las relaciones pueden ser unidireccionales o bidireccionales, pero la relación OneToMany unidireccional no se soporta directamente en Doctrine y requiere una tabla de unión.

Diferencia entre ManyToOne y OneToMany

ManyToOne y OneToMany son dos tipos de relaciones entre entidades, pero no son lo mismo. La diferencia radica en qué lado de la relación es el propietario y cómo se almacena la relación en la base de datos.

ManyToOne: En este tipo de relación, muchas entidades de un tipo están asociadas a una entidad de otro tipo. Por ejemplo, puedes tener muchos objetos Pedido asociados a un único objeto Cliente. En este caso, Pedido tendría una relación ManyToOne con Cliente.
El lado "muchos" (Pedido en este caso) es el propietario de la relación, lo que significa que la clave foránea que establece la relación se almacena en la tabla de la base de datos correspondiente a la entidad Pedido.

OneToMany: Este es el otro lado de una relación ManyToOne. En el ejemplo anterior, Cliente tendría una relación OneToMany con Pedido. Esto significa que un objeto Cliente puede estar asociado a muchos objetos Pedido. Sin embargo, a diferencia de ManyToOne, el lado "uno" (Cliente en este caso) no es el propietario de la relación. Esto significa que no hay ninguna clave foránea en la tabla de la base de datos correspondiente a Cliente que establezca la relación.
En resumen, ManyToOne y OneToMany se utilizan juntos para establecer una relación bidireccional entre dos tipos de entidades, pero el lado ManyToOne es el propietario de la relación y es el que tiene la clave foránea en la base de datos.

Trabajar con entidades

Trabajar con entidades me refiero a introducir, leer, actualizar y borrar datos en la tabla creada con las entidades.

Hidratación

Es el nombre que recibe el proceso de obtener un resultado final realizando una consulta a la base de datos y mapeando a un ResultSet. Los tipos de resultados que se devuelve en el proceso pueden ser:

- Entidades.

- Arrays estructurados → Array de toda la vida.
- Arrays escalares → que te devuelve un array, pero solo obtienes un resultado, es decir, te devuelve jugadores, pero obtienes el nombre de los jugadores.
- Variables simples.

Es un proceso que consume bastante, así que solo te interesa recuperar los datos que necesitas.

Consultas

En Doctrine 2 existen diferentes maneras de hacer consultas a la base de datos. Concretamente son 3 tipos:

- DQL → Lenguaje específico de consulta del dominio de Doctrine.
- QUERY BUILDER → Helper Class para la construcción de consultas.
- SQL Nativo → Uso de SQL propio del SGBD mediante NativeQuery o Query.

DQL

A favor:

- Es similar a SQL, pero tiene características propias.
- Gestiona objetos y propiedades en lugar de tablas y campos.
- Permite simplificar algunas construcciones de las consultas gracias al uso de metadatos.
- Es independiente al SGBD utilizado.

En contra:

- Diferencias con el SQL estàndard.
- No tiene toda la funcionalidad y optimizaciones de SQL específico de cada SGBD.
- Tiene limitaciones a la hora de implementar ciertas consultas.

Ejemplo de consulta DQL:

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

Query Builder

Clase creada para construir consultas SQL mediante una interfaz, a través de una API. Se crean las consultas mediante createQueryBuilder(), heredado del repositorio base de la entidad o desde entityManager. En la creación de QueryBuilder se debe especificar el parámetro (string), que es el alias de la entidad principal. Las consultas internamente hacen uso del **Prepared Statment** de SQL, por motivos de seguridad y rendimiento.

```
$qb = $entityRepo->createQueryBuilder("u");
// equivale a:
$qb = $entityManager->createQueryBuilder
$qb->select("u");
```

Se puede obtener una consulta DQL generada con el QueryBuilder mediante getDQL().

```
// Obtener DQL
$qb = $entityManager->createQueryBuilder();
$qb->select('p')->from('Cine\Entity\Pelicula','p');
$queryDQL = $qb->getDQL();

// Obtener SQL
query = $qb->getQuery();
$querySQL = $query->getSQL();
```

De una forma similar se puede obtener el SQL, con getSQL(). Ejemplo:

```
$qb = $entityManager->createQueryBuilder();
       $qb ->select("t.nombre", "COUNT(f.id) as num")
           ->from("Frases\Entity\Tema", "t")
           ->leftJoin("t.frases", "f")
           ->groupBy("t.id");
$query = $qb->getQuery();
$temes = $query->getResult();
```

SQL Nativo

Los resultados se mapean a entidades Doctrine mediante ResultSetMapBuilder. Se utiliza Doctrine ORM. Solo se soportan consultas SELECT. Es importante no mezclarlo con el concepto de Query, que es ejecutar SQL directo, como en el caso de DQL.

Ejemplo del Toni, copiado en clase (la estructura está bien, porque no peta, pero no se si devuelve resultado o no):

```
$sql = "SELECT * FROM tbl_temes t LEFT JOIN tbl_frases_temes ft ON t.id =
ft.tema_id";
$oRSMB = new \Doctrine\ORM\Query\ResultSetMappingBuilder($entityManager);
$oRSMB->addRootEntityFromClassMetadata('Frases\Entity\Tema', 't');
$oRSMB->addJoinedEntityFromClassMetadata('Frases\Entity\Frase',   'f',   't',
'frases', array('id' => 'frase_id'));

$qNativa = $entityManager->createNativeQuery($sql, $oRSMB);
$temes3 = $qNativa->getResult();
```

Ejemplos de GPT-4

Son ejemplos que me ha dado GPT-4 de cada uno.

DQL

```
// Sin pasar parámetros:
$query = $entityManager->createNativeQuery('SELECT p FROM App\Entity\Product p
WHERE p.price > 100');
$products = $query->getResult();

// Pasando parámetros:
$groupId = 123;

$dql = 'SELECT u FROM App\Entity\User u WHERE u.group = :group';
```

```php
$query = $entityManager->createQuery($dql);
$query->setParameter('group', $groupId);
$usersInGroup = $query->getResult();
```

Query Builder

```php
$queryBuilder = $entityManager->createQueryBuilder();
$queryBuilder->select('p');
->from('App\Entity\Product', 'p');
->where('p.price > :price');
->setParameter('price', 100);

$query = $queryBuilder->getQuery();
$products = $query->getResult();
```

SQL Nativo

```php
$sql = 'SELECT * FROM product WHERE price > 100';
$rsm = new Doctrine\ORM\Query\ResultSetMapping();
$rsm = addEntityResult('App\Entity\Product', 'p');
$rsm = addFieldResult('p', 'id', 'id');
$rsm = addFieldResult('p', 'name', 'name');
$rsm = addFieldResult('p', 'price', 'price');

$query = $entityManager->createNativeQuery($sql, $rsm);
$products = $query->getResult();
```

Acciones Básicas

Crear y añadir entidades a la base de datos

```php
use Frases\Entity\Autor;
use Frases\Entity\Tema;
// Crear un autor
$autor1 = new Autor();
$autor1->setNombre("Confuccio");
$autor1->setDescripcion("Pensador xinès");
$entityManager->persist($autor1);

// Crear temas, utilizando un array
$temes = array("Aprendre","Pensament","Vida","Justícia","Esperança");
       foreach ($temes as $t){
           $tema = new Tema();
           $tema->setNombre($t);
           $entityManager->persist($tema);
       }

// Crear frases utilizando temas ya existentes en la base de datos
$tema_pensament =
```

```
$entityManager->getRepository(Tema::class)->findOneBy(["nombre" =>
"Pensament"]);

$frase = new Frase();
$frase->setTexto("Cogito, ergo sum");
$frase->setAutor($entityManager->getRepository(Autor::class)->findOneBy(["nombre
" => "Descartes"]));
       $frase->addTema($tema_pensament);
$entityManager->persist($frase);

$entityManager->flush();
```

Leer / Obtener entidades de la base de datos

```
$autor_id = 1;
$autor_nombre = 'Confuccio';

// Obtener todas las entidades
$autors = $entityManager->getRepository('Frases\Entity\Autor')->findAll();
       foreach ($autors as $a){
           echo "- " . $a->getNombre() . "<br>";
       }

// Obtener entidad por id
$autor = $entityManager->getRepository(Autor::class)->find($autor_id);

// Obtener entidad por un campo, en este caso por nombre
$autor2 = $entityManager->getRepository(Autor::class)->findOneBy(['nombre' =>
$autor_nombre]);
```

Actualizar entidades

```
$autor_id = 1;
$autor = $entityManager->getRepository(Autor::class)->find($autor_id);
       $autor->setNombre("Hugo");
       $entityManager->persist($autor);

$entityManager->flush();
```

Eliminar entidades

Se que no lo hemos hecho, pero por si acaso. Se lo he pedido al GPT-4.

```
$prod_id = 1;
$prod = $entityManager->getRepository(Product::class)->find($prod_id);
if ($prod != null) {
      $entityManager->remove($prod);
      $entityManager->flush();
}
```

Repositorios

Cuando generamos una entidad, Doctrine proporciona un repositorio base por defecto que contiene una serie de métodos para operar:

- find(id): Devuelve una entidad con el id, o null si no existe.
- findAll(): Devuelve un array con todas las entidades del repositorio.
- findBy(array(criterios) [, array (a ordenación)]): Devuelve una matriz con todas las entidades que cumplan los criterios especificados en el primer parámetro, y ordenados por el segundo parámetro (opcional).
- findOneBy(array(criterios)): Similar a findBy() pero devuelve solo un elemento o null.

Algunos métodos se pueden usar de forma abreviada. Ejemplo:

```
findByTitulo('valor'), findByOneTitulo('valor');
// equivale a:
findBy('Titulo'=>'valor'), findOneBy('Titulo'=>'valor');
```

Esto se debe a que Doctrine usa el método mágico __call() de PHP, que hace a Doctrine localizar el nombre del método de la clase. Por este motivo, no podemos usar call.
El repositorio base de una entidad puede ser extendido para proporcionar métodos específicos para las consultas del usuario. Este repositorio te permite optimizar la obtención de resultados en las consultas, teniendo mayor control en la hidratación de datos y especificar acciones que no se harían de forma automática. Doctrine permite especificar la clase que usaremos como repositorio, agregando a la anotación: `@Entity(repositoryClass="Frases\Repository\TemaRepository")`

Hecho esto, hay que poner en la consola el siguiente comando:
php vendor/bin/doctrine.php orm:generate-repositories /src
Hecho esto, lo suyo es que vayas al composer.json y agregues el espacio de nombres, porque sino no te va a funcionar. Si por ejemplo has puesto el espacio de nombres /Frases/Repository tienes que agregar, dentro del PSR-4, lo siguiente:

```
"Frases\\Repository\\" : "src/Frases/Repository"
```

Comandos consola

IMPORTANTE: todos se deben ejecutar en la carpeta raíz del proyecto, sino no funcionaran.

```
// INSTALAR DOCTRINE
// Si ya tienes el composer.json...
composer install
// Si no lo tienes y lo quieres generar, para no escribirlo a mano...
composer init
      // En caso de hacerlo así, poner los siguientes comandos...
      [ENTER]
      [Poner descripción]
      [Poner autor]
```

```
      [ENTER]
      project
      [ENTER]
      y
      orm
      0
      2.8 // Puedes poner la versión que quieras. Si quieres la última
darle a [ENTER] o poner en el composer.json un *
      n
      src/
      y
      y

/* Si lo haces así, no tendrás configurado el namespace, tendrás que ir
a composer.json y entonces cambiar dentro del apartado PSR-4 poner el
namespace que quieras. IMPORTANTE: Si modificas el composer.json irte a
otra pestaña volver y guardar. Esto hay que hacerlo porque el Eclipse es
una mierda. Ej. de namespace (puedes poner varios): */
"Frases\\Entity\\" : "src/Frases/Entity",
"Frases\\Repository\\" : "src/Frases/Repository"

// Si quieres saber si todo se ha instalado correctamente...
php vendor/doctrine/orm/bin/doctrine.php

// ACTUALIZAR DOCTRINE
// Si borras la carpeta vendor y el composer.lock...
composer install
// Si modificas el composer.json y quieres actualizar el proyecto...
composer update

// ENTIDADES
// Generar las entidades...
php vendor/doctrine/orm/bin/doctrine.php orm:generate:entities src/

// BASE DE DATOS (si por lo que falla, usar --force)
// Generar base de datos, con las entidades definidas...
php vendor/doctrine/orm/bin/doctrine.php orm:schema-tool:create
// Actualizar la base de datos, habiendo modificado las entidades
definidas...
php vendor/doctrine/orm/bin/doctrine.php orm:schema-tool:update
// Borrar la base de datos
php vendor/doctrine/orm/bin/doctrine.php orm:schema-tool:drop --force

// REPOSITORIOS
// Generar los repositorios definidos en las entidades...
php vendor/bin/doctrine.php orm:generate-repositories /src
```