PDO --> capa entre aplicacion y base de datos

CORE --> provee la interface, que clases y metodos tiene que tener la base de datos
DRIVER --> Acceso a cada base de datos particular (El driver lo proporciona el fabricante)

para inicializar:
$pdo = new PDO (
datos (Ej. "mysql:host='localhost'; dbname='testdb'...")
);

DOCTRINE

Doctrine es un ORM

La idea és fer algo per no preocuparse per com funciona cada base de dades i no complicar-se la vida.
Jo dir: Graba, i que ell s'encarregui de grabar com sigui

Doctrine és un conjunt de llibreries

patrons --> solucions facils a problemes cotidians
entitat --> allo que jo vull persistir
Entity Manager --> carrega la informació i hidrata les taules hidratant el objecte amb tot el que necessiti
Unity of Work --> Nomes funciona si totes les transaccions han funcionat, sino, rollback

INSTAL·LACIÓ
1. Instalar composer (un gestor de paquets)
2. Un cop instalat, el movem al local/bin/composer

COMANDOS
$ composer init
$ composer install
$ composer update

con las entidades creadas, para hacer los getters / setters --> php vendor/doctrine/orm/bin/doctrine.php orm:generate:entities src/

crear el esquema en la base de datos --> php vendor/doctrine/orm/bin/doctrine.php orm:schema-tool:create (aparte del create, tambien hay el update y el drop macho que te lo tengo que decir todo. Tambien puedes enchufar por ahi el --force si algo t esta tocando la polla))

para generar los repositorios --> php vendor/doctrine/orm/bin/doctrine.php orm:generate-repositories /src

(recuerda que para lo de repositorios tienes que añadir en cada entidad lo del repositorio en el entity poner lo de repositoryClass="Frases\Repository\AutorRepository" y en el composer poner esto: "Frases\\Repository\\" : "src/Frases/Repository")

RELACIONS DE CLASSES A DOCTRINE

Quan fas una Entitat a doctrine (una classe), tens varies opcions a la hora de fer relacions entre classes:

(Los getters y setters estan raros, lo se, pero los he dejado pq igual es otra forma de hacerlo e igual te lo pone por ahi tipo: esto esta bien? se puede hacer asi? (SI SIEMPRE SE TIENE QUE HACER ASI DECIDMELO CABRONES)).

==========
1. OneToOne
==========
Cada entitat principal només pot tenir una associada. Exemple: Un usuari només pot tenir un perfil

```
<?php
use Doctrine\ORM\Mapping as ORM;

/**
 * @Entity
 * @Table(name="usuari")
 */
class Usuari {
      // ...

      /**
      * @OneToOne(targetEntity="Perfil", cascade={"persist", "remove"})
      * @JoinColumn(name="perfil_id", referencedColumnName="id")
      */
      private $perfil;

      // ...

      public function getPerfil(): ?Perfil
      {
      return $this->perfil;
      }

      public function setPerfil(Perfil $perfil): self
      {
      $this->perfil = $perfil;

      return $this;
      }
}

/**
 * @Entity
```

```
 * @Table(name="perfil")
 */
class Perfil {
      // ...

      /**
      * @OneToOne(targetEntity="Usuari", mappedBy="perfil")
      */
      private $usuari;

      // ...

      public function getUsuari(): ?Usuari
      {
      return $this->usuari;
      }

      public function setUsuari(Usuari $usuari): self
      {
      $this->usuari = $usuari;

      return $this;
      }
}
```

==========

2. OneToMany

==========

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

```
<?php
use Doctrine\ORM\Mapping as ORM;

/**
 * @Entity
 * @Table(name="autor")
 */
class Autor {
      // ...

      /**
      * @OneToMany(targetEntity="Libro", mappedBy="autor",
cascade={"persist", "remove"})
      */
      private $libros;

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

      // ...

      public function getLibros(): \Doctrine\Common\Collections\Collection
```

```
	{
	return $this->libros;
	}

	public function addLibro(Libro $libro): self
	{
	if (!$this->libros->contains($libro)) {
		$this->libros[] = $libro;
		$libro->setAutor($this);
	}

	return $this;
	}

	public function removeLibro(Libro $libro): self
	{
	if ($this->libros->contains($libro)) {
		$this->libros->removeElement($libro);
		// set the owning side to null
		if ($libro->getAutor() === $this) {
		$libro->setAutor(null);
		}
	}

	return $this;
	}
}

/**
 * @Entity
 * @Table(name="libro")
 */
class Libro {
	// ...

	/**
	* @ManyToOne(targetEntity="Autor", inversedBy="libros")
	* @JoinColumn(name="autor_id", referencedColumnName="id")
	*/
	private $autor;

	// ...

	public function getAutor(): ?Autor
	{
	return $this->autor;
	}

	public function setAutor(?Autor $autor): self
	{
	$this->autor = $autor;

	return $this;
```

```
      }
}
?>
```

```
================================================================================
3. ManyToOne
================================================================================
Varias entidades pueden estar asociadas a una entidad principal. Ejemplo:
Varios libros pueden estar asociados a un solo autor.

<?php
use Doctrine\ORM\Mapping as ORM;

/**
 * @Entity
 * @Table(name="libro")
 */
class Libro {
      // ...

      /**
      * @ManyToOne(targetEntity="Autor", inversedBy="libros")
      * @JoinColumn(name="autor_id", referencedColumnName="id")
      */
      private $autor;

      // ...

      public function getAutor(): ?Autor
      {
      return $this->autor;
      }

      public function setAutor(?Autor $autor): self
      {
      $this->autor = $autor;

      return $this;
      }
}

/**
 * @Entity
 * @Table(name="autor")
 */
class Autor {
      // ...

      /**
      * @OneToMany(targetEntity="Libro", mappedBy="autor",
cascade={"persist", "remove"})
      */
      private $libros;
```

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

      // ...

      public function getLibros(): \Doctrine\Common\Collections\Collection
      {
      return $this->libros;
      }

      public function addLibro(Libro $libro): self
      {
      if (!$this->libros->contains($libro)) {
            $this->libros[] = $libro;
            $libro->setAutor($this);
      }

      return $this;
      }

      public function removeLibro(Libro $libro): self
      {
      if ($this->libros->contains($libro)) {
            $this->libros->removeElement($libro);
            // set the owning side to null
            if ($libro->getAutor() === $this) {
            $libro->setAutor(null);
            }
      }

      return $this;
      }
}
?>
```

==========

4. ManyToMany

==========

Varias entidades pueden estar asociadas a varias entidades. Ejemplo: Un estudiante puede estar matriculado en varios cursos y un curso puede tener varios estudiantes matriculados.

```
<?php
use Doctrine\ORM\Mapping as ORM;

/**
 * @Entity
 * @Table(name="estudiante")
 */
class Estudiante {
      // ...
```

```
	/**
	* @ManyToMany(targetEntity="Curso", inversedBy="estudiantes")
	* @JoinTable(name="estudiantes_cursos")
	*/
	private $cursos;

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

	// ...

	public function getCursos(): \Doctrine\Common\Collections\Collection
	{
	return $this->cursos;
	}

	public function addCurso(Curso $curso): self
	{
	if (!$this->cursos->contains($curso)) {
		$this->cursos[] = $curso;
		$curso->addEstudiante($this);
	}

	return $this;
	}

	public function removeCurso(Curso $curso): self
	{
	if ($this->cursos->contains($curso)) {
		$this->cursos->removeElement($curso);
		$curso->removeEstudiante($this);
	}

	return $this;
	}
}

/**
 * @Entity
 * @Table(name="curso")
 */
class Curso {
	// ...

	/**
	* @ManyToMany(targetEntity="Estudiante", mappedBy="cursos")
	*/
	private $estudiantes;

	public function __construct() {
```

```
      $this->estudiantes = new
\Doctrine\Common\Collections\ArrayCollection();
      }

      // ...

      public function getEstudiantes():
\Doctrine\Common\Collections\Collection
      {
      return $this->estudiantes;
      }

      public function addEstudiante(Estudiante $estudiante): self
      {
      if (!$this->estudiantes->contains($estudiante)) {
            $this->estudiantes[] = $estudiante;
            $estudiante->addCurso($this);
      }

      return $this;
      }

      public function removeEstudiante(Estudiante $estudiante): self
      {
      if ($this->estudiantes->contains($estudiante)) {
            $this->estudiantes->removeElement($estudiante);
            $estudiante->removeCurso($this);
      }

      return $this;
      }
}
?>
```

==========

COSITAS DEL @

@Entity: Define que la clase es una entidad que se puede persistir en la base de datos.

-repositoryClass: Define la clase del repositorio personalizado para esta entidad. -->
@Entity(repositoryClass="Frases\Repository\AutorRepository")

@Table: Define la tabla de la base de datos asociada a la entidad.

-name: Nombre de la tabla. --> @Table(name="autores")
-schema: Esquema de la tabla. --> @Table(name="autores", schema="public")
-indexes: Índices de la tabla. --> @Table(name="autores", indexes={@Index(name="nombre_dni_idx", columns={"nombre", "dni"})})

-uniqueConstraints: Restricciones únicas de la tabla. --> @Table(name="autores", uniqueConstraints={@UniqueConstraint(name="dni_unico", columns={"dni"})})
-options: Opciones adicionales de la tabla. --> @Table(name="autores", options={"comment"="Tabla de autores"})

//Estas dudo que las vaya a pedir (ademas estas me las ha generado automaticamente el github al entender el contexto de lo que estoy haciendo, x eso digo que dudo que las pida)
-inheritanceType: Tipo de herencia de la tabla. --> @Table(name="autores", inheritanceType="JOINED")
-discriminatorColumn: Columna discriminadora de la tabla. --> @Table(name="autores", discriminatorColumn=@DiscriminatorColumn(name="tipo", type="string"))
-discriminatorMap: Mapa discriminador de la tabla. --> @Table(name="autores", discriminatorMap=@DiscriminatorMap({"autor"="Autor", "editor"="Editor"}))
-catalog: Catálogo de la tabla. --> @Table(name="autores", catalog="biblioteca")
-primaryKey: Clave primaria de la tabla. --> @Table(name="autores", primaryKey={@PrimaryKey(name="id")})

@Column: Define una columna en la tabla.

-type: Tipo de la columna. --> @Column(type="string") || @Column(type="integer") || @Column(type="datetime") || @Column(type="text") || @Column(type="boolean") || @Column(type="float") || @Column(type="decimal") || @Column(type="array") no se si hay mas bro, ya hay que tener mala hostia para pedir uno que no sea de estos loko no m ralles
-length: Longitud de la columna. --> @Column(type="string", length=255)
-unique: Define si la columna es única. --> @Column(type="string", unique=true)
-nullable: Define si la columna puede ser nula. --> @Column(type="string", nullable=true)
-options: Opciones adicionales de la columna. --> @Column(type="string", options={"comment"="Nombre del autor"})
-name: Nombre de la columna. --> @Column(name="nombre", type="string")

//Estas son las del colegon, también dudo que las pida
-precision: Precisión de la columna. --> @Column(type="decimal", precision=10, scale=2)
-scale: Escala de la columna. --> @Column(type="decimal", precision=10, scale=2)
-columnDefinition: Definición de la columna. --> @Column(type="string", columnDefinition="VARCHAR(255) NOT NULL")

@Id: Define la columna como una clave primaria.

@GeneratedValue: Define la estrategia de generación de la clave primaria.
-strategy: Estrategia de generación (AUTO, SEQUENCE, IDENTITY, NONE, UUID, CUSTOM). --> @GeneratedValue(strategy="AUTO")

//Los del experto en todo, el que todo lo sabe, el todopoderoso, el sin preguntas, el que no necesita ayuda, el que ya lo sabe todo, el que no necesita aprender nada, el que no necesita estudiar, el que sabe lo que sabe, o maestro, el que sabe cositas

-generator: Nombre del generador de secuencias. --> @GeneratedValue(strategy="SEQUENCE", generator="secuencia_id")
Esta opción se utiliza cuando la estrategia de generación es "SEQUENCE". Especifica el nombre del generador de secuencias que se utilizará.
En tu ejemplo, @GeneratedValue(strategy="SEQUENCE", generator="secuencia_id"), secuencia_id es el nombre del generador de secuencias que se utilizará
para generar valores para la clave primaria.

-allocationSize: Tamaño de la asignación. --> @GeneratedValue(strategy="SEQUENCE", generator="secuencia_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.
En tu ejemplo, @GeneratedValue(strategy="SEQUENCE", generator="secuencia_id", allocationSize=1), el tamaño de la asignación es 1, lo que significa que se reservará una secuencia a la vez.

@OneToOne: Define una relación uno a uno.

-targetEntity: Clase de la entidad objetivo. --> @OneToOne(targetEntity="Perfil")
-mappedBy: Define el lado propietario de la relación. --> @OneToOne(targetEntity="Perfil", mappedBy="usuario")
-cascade: Define las operaciones en cascada (persist, remove, merge, detach, all). --> @OneToOne(targetEntity="Perfil", cascade={"persist", "remove"}) || @OneToOne(targetEntity="Perfil", cascade={"all"})
-orphanRemoval: Define si las entidades huérfanas se deben eliminar. --> @OneToOne(targetEntity="Perfil", orphanRemoval=true)
-inversedBy: Define el lado no propietario de la relación. --> @OneToOne(targetEntity="Perfil", inversedBy="usuario")

//Estas son las del colegon, también dudo que las pida
-fetch: Define cómo se obtiene la entidad relacionada (EAGER, LAZY). --> @OneToOne(targetEntity="Perfil", fetch="EAGER")

@OneToMany: Define una relación uno a muchos.
(LAS MISMAS QUE ONETOONE CABRON)

-targetEntity: Clase de la entidad objetivo. -->
@OneToMany(targetEntity="Libro")
-mappedBy: Define el lado propietario de la relación. -->
@OneToMany(targetEntity="Libro", mappedBy="autor")
-cascade: Define las operaciones en cascada (persist, remove, merge, detach, all). --> @OneToMany(targetEntity="Libro", cascade={"persist", "remove"}) || @OneToMany(targetEntity="Libro", cascade={"all"})
-orphanRemoval: Define si las entidades huérfanas se deben eliminar. --> @OneToMany(targetEntity="Libro", orphanRemoval=true)

//Estas son las del colegon, también dudo que las pida
-fetch: Define cómo se obtiene la entidad relacionada (EAGER, LAZY). --> @OneToMany(targetEntity="Libro", fetch="EAGER")

@ManyToOne: Define una relación muchos a uno.

-targetEntity: Clase de la entidad objetivo. -->
@ManyToOne(targetEntity="Autor")
-cascade: Define las operaciones en cascada (persist, remove, merge, detach, all). --> @ManyToOne(targetEntity="Autor", cascade={"persist", "remove"})
-fetch: Define cómo se obtiene la entidad relacionada (EAGER, LAZY). --> @ManyToOne(targetEntity="Autor", fetch="EAGER")
-joinColumns: Define las columnas de unión. -->
@ManyToOne(targetEntity="Autor", joinColumns={@JoinColumn(name="autor_id", referencedColumnName="id")})
-inversedBy: Define el lado no propietario de la relación. -->
@ManyToOne(targetEntity="Autor", inversedBy="libros")

BRO creo que este no tiene orphanRemoval pero no estoy seguro OKEY?! QUEDA CLARO?

@ManyToMany: Define una relación muchos a muchos.

-targetEntity: Clase de la entidad objetivo. -->
@ManyToMany(targetEntity="Categoria")
-mappedBy: Define el lado propietario de la relación. -->
@ManyToMany(targetEntity="Categoria", mappedBy="libros")
-cascade: Define las operaciones en cascada (persist, remove, merge, detach, all). --> @ManyToMany(targetEntity="Categoria", cascade={"persist", "remove"})
- fetch: Define cómo se obtiene la entidad relacionada (EAGER, LAZY). --> @ManyToMany(targetEntity="Categoria", fetch="EAGER")
-joinTable: Define la tabla de unión. -->
@ManyToMany(targetEntity="Categoria", joinTable=@JoinTable(name="libros_categorias", joinColumns={@JoinColumn(name="libro_id", referencedColumnName="id")}, inverseJoinColumns={@JoinColumn(name="categoria_id", referencedColumnName="id")})

```
       -inversedBy: Define el lado no propietario de la relación. -->
@ManyToMany(targetEntity="Categoria", inversedBy="libros")

@JoinColumn: Define la columna de unión para una relación.

       -name: Nombre de la columna de unión. -->
@JoinColumn(name="autor_id")
       -referencedColumnName: Nombre de la columna referenciada. -->
@JoinColumn(name="autor_id", referencedColumnName="id")
       -unique: Define si la columna de unión es única. -->
@JoinColumn(name="autor_id", unique=true)
       -nullable: Define si la columna de unión puede ser nula. -->
@JoinColumn(name="autor_id", nullable=true)
       -onDelete: Acción a realizar cuando se elimina la entidad
referenciada (SET NULL, CASCADE). --> @JoinColumn(name="autor_id",
onDelete="SET NULL")
       -columnDefinition: Definición de la columna de unión. -->
@JoinColumn(name="autor_id", columnDefinition="INT NOT NULL")
       -table: Nombre de la tabla de unión. --> @JoinColumn(name="autor_id",
table="libros_autores")


@JoinTable: Define la tabla de unión para una relación ManyToMany.

       -name: Nombre de la tabla de unión. -->
@JoinTable(name="libros_categorias")
       -joinColumns: Columnas de unión. -->
@JoinTable(name="libros_categorias",
joinColumns={@JoinColumn(name="libro_id", referencedColumnName="id")})
       -inverseJoinColumns: Columnas de unión inversa. -->
@JoinTable(name="libros_categorias",
inverseJoinColumns={@JoinColumn(name="categoria_id",
referencedColumnName="id")})
       -uniqueConstraints: Restricciones únicas de la tabla de unión. -->
@JoinTable(name="libros_categorias",
uniqueConstraints={@UniqueConstraint(name="libro_categoria_unico",
columns={"libro_id", "categoria_id"})})
       -indexes: Índices de la tabla de unión. -->
@JoinTable(name="libros_categorias",
indexes={@Index(name="libro_categoria_idx", columns={"libro_id",
"categoria_id"})})


@SequenceGenerator: Define un generador de secuencias para una columna.

       -sequenceName: Nombre de la secuencia. -->
@SequenceGenerator(sequenceName="secuencia_id")
       -allocationSize: Tamaño de la asignación. -->
@SequenceGenerator(sequenceName="secuencia_id", allocationSize=1)
       -initialValue: Valor inicial. -->
@SequenceGenerator(sequenceName="secuencia_id", initialValue=1)
```

@OrderBy: Define el orden de las entidades en una colección.
	-value: Orden de las entidades. --> @OrderBy({"nombre" = "ASC"})

@Version: Define una columna de versión para el control de concurrencia optimista.

@InheritanceType: Define el tipo de herencia de la entidad.
	-value: Tipo de herencia (SINGLE_TABLE, JOINED, TABLE_PER_CLASS). --> @InheritanceType("JOINED")

@DiscriminatorColumn: Define la columna discriminadora para la herencia de entidad.

	-name: Nombre de la columna discriminadora. --> @DiscriminatorColumn(name="tipo", type="string")
	-type: Tipo de la columna discriminadora. --> @DiscriminatorColumn(name="tipo", type="string")

@DiscriminatorMap: Define el mapa discriminador para la herencia de entidad.

	-value: Mapa discriminador. --> @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.
	-name: Nombre de la restricción única. --> @UniqueConstraint(name="dni_unico", columns={"dni"})
	-columns: Columnas de la restricción única. --> @UniqueConstraint(name="dni_unico", columns={"dni"})

@Index: Define un índice para una tabla.
	-name: Nombre del índice. --> @Index(name="nombre_dni_idx", columns={"nombre", "dni"})
	-columns: Columnas del índice. --> @Index(name="nombre_dni_idx", columns={"nombre", "dni"})

@AttributeOverrides y @AttributeOverride: Permiten anular los detalles de mapeo de una propiedad incrustada.

@AssociationOverrides y @AssociationOverride: Permiten anular los detalles de mapeo de una asociación 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.

==========

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.

con relacion bidireccional:

```
<?php

class Cliente {
      // ...

      /**
      * @OneToMany(targetEntity="Pedido", mappedBy="cliente")
      */
      private $pedidos;

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

```
        // ...

        public function getPedidos(): \Doctrine\Common\Collections\Collection
        {
        return $this->pedidos;
        }

        public function addPedido(Pedido $pedido): self
        {
        if (!$this->pedidos->contains($pedido)) {
              $this->pedidos[] = $pedido;
              $pedido->setCliente($this);
        }

        return $this;
        }

        public function removePedido(Pedido $pedido): self
        {
        if ($this->pedidos->contains($pedido)) {
              $this->pedidos->removeElement($pedido);
              // set the owning side to null
              if ($pedido->getCliente() === $this) {
              $pedido->setCliente(null);
              }
        }

        return $this;
        }
}

class Pedido {
        // ...

        /**
        * @ManyToOne(targetEntity="Cliente", inversedBy="pedidos")
        * @JoinColumn(name="cliente_id", referencedColumnName="id")
        */
        private $cliente;

        // ...

        public function getCliente(): ?Cliente
        {
        return $this->cliente;
        }

        public function setCliente(?Cliente $cliente): self
        {
        $this->cliente = $cliente;

        return $this;
        }
```

```
}
?>
```

Con relacion unidireccional:

Puedes definir una relación ManyToOne sin definir el lado OneToMany correspondiente, y viceversa. Esto se conoce como una relación unidireccional.
Si solo defines el lado ManyToOne, tendrás una relación de muchos a uno desde la entidad del lado "muchos" a la entidad del lado "uno",
pero no podrás navegar en la dirección opuesta.

```
class Pedido {
	// ...

	/**
	* @ManyToOne(targetEntity="Cliente")
	* @JoinColumn(name="cliente_id", referencedColumnName="id")
	*/
	private $cliente;

	// ...
}
```

En este código, un Pedido puede estar asociado a un Cliente, pero no puedes obtener todos los Pedidos de un Cliente directamente desde el objeto Cliente.
De manera similar, puedes definir una relación OneToMany sin definir el lado ManyToOne correspondiente.
Sin embargo, esto es menos común porque el lado "uno" no puede ser el propietario de la relación (es decir, no puede tener la clave foránea en su tabla).
Esto significa que necesitarías una tabla de unión para almacenar la relación, lo cual es menos eficiente y más complicado de manejar que una simple clave foránea.
En resumen, aunque es posible definir solo uno de los lados de una relación ManyToOne/OneToMany,
generalmente es más útil y eficiente definir ambos lados y hacer que la relación sea bidireccional.

==========

REGLAS DE BIDIRECCIONAL // UNIDIRECCIONAL

En Doctrine, todas las relaciones (OneToOne, OneToMany, ManyToOne, ManyToMany) pueden ser unidireccionales o bidireccionales.
Sin embargo, hay algunas consideraciones a tener en cuenta:

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

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

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

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.

/*

ALEJANDRO! PARA SABER SI HACER UNIDIRECCIONAL O BIDIRECCIONAL

La elección entre una relación unidireccional y bidireccional en Doctrine depende en gran medida de las necesidades de tu aplicación y de cómo planeas utilizar los datos.

Si solo necesitas navegar desde Autor a Perfil (por ejemplo, si solo necesitas obtener el perfil de un autor, pero nunca necesitas obtener el autor de un perfil), entonces una relación unidireccional sería suficiente.

Por otro lado, si necesitas poder navegar en ambas direcciones (por ejemplo, si necesitas obtener el perfil de un autor y también necesitas obtener el autor de un perfil), entonces necesitarías una relación bidireccional.

En tu caso, si consideras que en algún momento podrías necesitar obtener el autor a partir de un perfil, entonces sería mejor optar por una relación bidireccional. Sin embargo, si estás seguro de que solo necesitarás obtener el perfil a partir del autor, entonces una relación unidireccional sería suficiente.

Además, ten en cuenta que las relaciones bidireccionales son un poco más complejas de manejar que las unidireccionales, ya que debes asegurarte de mantener ambos lados de la relación sincronizados. Por lo tanto, si no necesitas la bidireccionalidad, podría ser más sencillo optar por una relación unidireccional.