

# TEMA4

## CREACIÓN DE UNA BASE DE DATOS

# INTRODUCCIÓN

- Symfony incluye Doctrine, que es una ORM (object relational mapping) para facilitar el acceso a datos tanto relacionales como noSQL, siempre que se ajusten al paradigma de la orientación a objetos.
- Con Doctrine podemos tener ya los datos dinámicamente almacenados en nuestra base de datos y manipularlos.
- ORM nos permite abstraernos del motor de base de datos que estemos usando para el proyecto y nos permitirá, con solo un poco de configuración, cambiar toda nuestra aplicación por ejemplo de una base de datos MySQL a SQLSERVER o a cualquier otra soportada por el framework Doctrine.

# COMANDOS NECESARIOS

Lo primero que hay que hacer es instalar todos los módulos que vamos a usar con los comandos:

## 1. **composer require maker --dev :**

Ayuda a generar código creando comandos vacíos, controladores, clases de formulario, etc, para no tener que escribir código repetitivo

## 2. **composer require doctrine :**

Instala el soporte ORM de doctrine

# COMANDOS NECESARIOS

- Con esto ya tenemos las cosas básicas, para crear nuestras primeras entidades, las cuales se guardan en el directorio: `src/entity`.
- A partir de ahora, nos olvidamos de trabajar directamente los ficheros de entidades como en versiones anteriores.
- Estos ficheros podemos generarlos con el maker de Symfony cuyo formato es:
  - 3. `php bin/console make:entity nombre-entidad`
- Muy pocas veces va a ser necesario programar estos ficheros directamente, aparte de que sólo tienen que usarse para los getters, setters y colecciones.

# FICHERO .ENV

Como hemos visto en la estructura del proyecto symfony, existe un fichero llamado **.env** , este es un fichero donde tenemos la configuración básica de nuestra aplicación.(Entorno, secreto, conexión a base de datos)

```
# the latter taking precedence over the former:
#
# * .env                contains default values for the environment variables needed by the app
# * .env.local          uncommitted file with local overrides
# * .env.$APP_ENV       committed environment-specific defaults
# * .env.$APP_ENV.local uncommitted environment-specific overrides
#
# Real environment variables win over .env files.
#
# DO NOT DEFINE PRODUCTION SECRETS IN THIS FILE NOR IN ANY OTHER COMMITTED FILES.
# https://symfony.com/doc/current/configuration/secrets.html
#
# Run "composer dump-env prod" to compile .env files for production use (requires symfony/flex >=1.2).
# https://symfony.com/doc/current/best\_practices.html#use-environment-variables-for-infrastructure-configuration

###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=d06158e5ae23182a09245bfb7a400a8f
###< symfony/framework-bundle ###

###> doctrine/doctrine-bundle ###
# Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html
# IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
#
# DATABASE_URL="sqlite:///kernel.project_dir/var/data.db"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=16&charset=utf8"
###< doctrine/doctrine-bundle ###

###> symfony/messenger ###
# Choose one of the transports below
# MESSENGER_TRANSPORT_DSN=amqp://quest:quest@localhost:5672/%2f/messages
# MESSENGER_TRANSPORT_DSN=redis://localhost:6379/messages
MESSENGER_TRANSPORT_DSN=doctrine://default?auto_setup=0
```

# FICHERO .ENV

Como podemos ver en el fichero env, nos indica que estamos en el entorno de desarrollo[app\_env:dev] una vez esté terminado y se suba para que los usuarios interactúe con el se cambia dev por prod.

Otra línea de gran interés para nosotros es DATABASE\_URL, nos indica el motor de la base de datos en nuestro caso mysql, a continuación pondríamos el usuario (root),seguido de la contraseña, el nombre de la base de datos , y ya estaría configurada la BBDD, solo hay que salvar los cambios y podríamos inicializar la BBDD.

Tendríamos que comentar la línea que venía activa por defecto, y activar:

➤ **DATABASE\_URL=mysql://root@127.0.0.1:3306/nombre-bbdd**

```
###> doctrine/doctrine-bundle ###
# Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html
# IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
#
# DATABASE_URL="sqlite:///var/data.db"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"

DATABASE_URL=mysql://root@127.0.0.1:3306/proyector1

# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"

#DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=16&charset=utf8"

###< doctrine/doctrine-bundle ###

###> symfony/messenger ###
# Choose one of the transports below
# MESSENGER_TRANSPORT_DSN=amqp://quest:quest@localhost:5672/%2f/messages
# MESSENGER_TRANSPORT_DSN=redis://localhost:6379/messages
MESSENGER_TRANSPORT_DSN=doctrine://default?auto_setup=0
###< symfony/messenger ###
```



# CREAR EL ESPACIO DE TABLAS PARA MI BBDD

Para inicializar la BBDD, con el cmd voy al raíz de mi proyecto, y desde ahí escribo el comando:

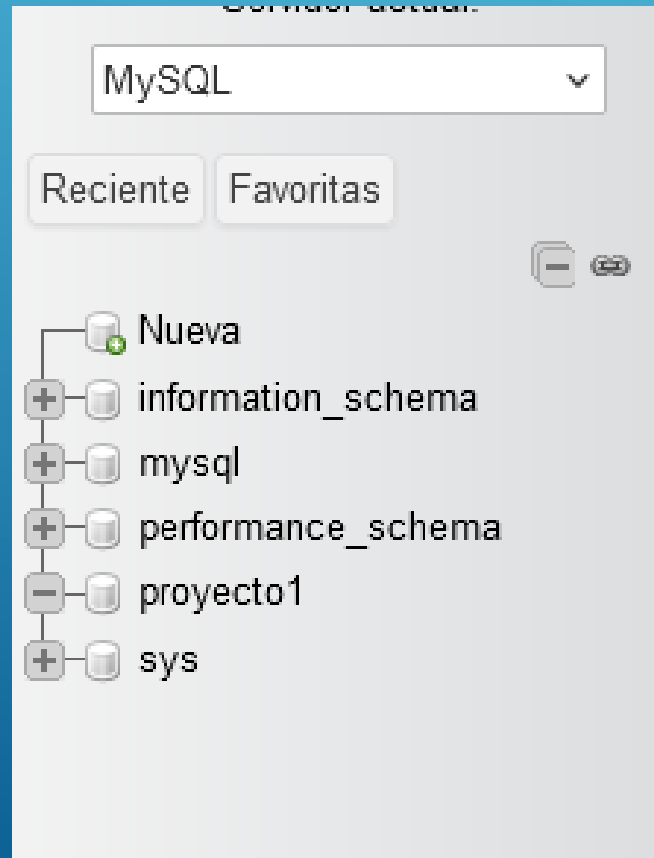
**php bin/console doctrine:database:create**

Con esta instrucción se comprueba que en el fichero .env, esta bien configurado y se creará el espacio donde se alojará la bbdd.

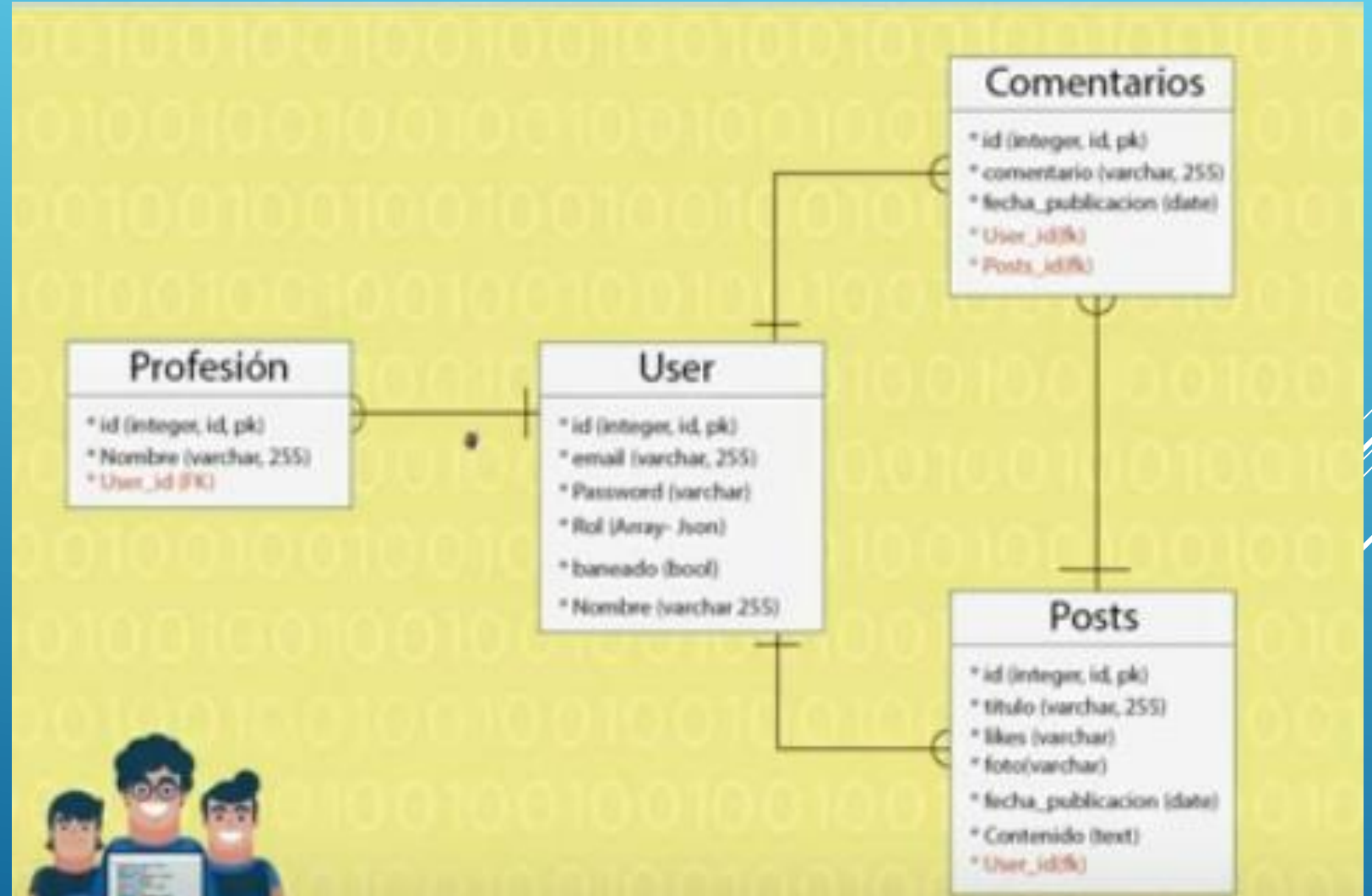
```
C:\wamp64\www\miAppSymfony>
C:\wamp64\www\miAppSymfony>php bin/console doctrine:database:create
Created database `proyecto1` for connection named default
C:\wamp64\www\miAppSymfony>
```

# CREAR EL ESPACIO DE TABLAS PARA MI BBDD

Si vamos a phpMyadmin vemos el tablesaces aún vacío :



PARA CREAR LAS TABLAS NOS VAMOS A BASAR EN EL SIGUIENTE MER:



# CREANDO LAS ENTIDADES

- Como he dicho al principio el ORM que utiliza symfony es Doctrine que me sirve para comunicar nuestra aplicación con la BBDD.
- La bbdd necesita de unas entidades para transformarlas en tablas físicas, en este framework una entidad no es mas que una clase php.
- Para crear la tabla post, por ejemplo, voy de nuevo al raíz de proyecto . Y ejecuto:

➤ **php bin/console make:entity**

A continuación symfony me pregunta el nombre de mi entidad que la llamare como quiero que se llame la tabla, en este caso : Posts

Es muy importante que se **empiece siempre con mayúsculas** para evitar errores tediosos en un futuro (tendría conflicto con el nombre de la clase que siempre se genera en mayúsculas).

# CREANDO LAS ENTIDADES

Una vez se empieza a ejecutar me indica donde se va a crear esa entidad, su nombre que será Posts.php, y el nombre de su repositorio, que será Postsrepository.php.

```
C:\wamp64\www\miAppSymfony2>php bin/console make:entity

Class name of the entity to create or update (e.g. TinyKangaroo):
> Post

Add the ability to broadcast entity updates using Symfony UX Turbo? (yes/no) [no]:
> no

created: src/Entity/Post.php
created: src/Repository/PostRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.
```

# CREANDO LAS ENTIDADES

A continuación vamos a describir sus campos.

Hay que meter todos menos el id, porque este se genera automáticamente y de auto-incremento y lo pone de clave primaria

En el resto le tengo que ir contestando a las preguntas que son:

- El nombre que le doy

- El tipo de datos que va a albergar, si no sabemos qué tipos soporta pongo **un** ? Y me dá todas las posibilidades que hay.

- La longitud que le doy el número de caracteres que necesito

- Si puede ser o no nulo, por defecto es no nulo.

Cuando no quiera más campos, a la pregunta si añadimos más, le damos enter y se hace automáticamente la clase.

# ENTIDAD POST

```
New property name (press <return> to stop adding fields):
```

```
> titulo
```

```
Field type (enter ? to see all types) [string]:
```

```
>
```

```
Field length [255]:
```

```
>
```

```
Can this field be null in the database (nullable) (yes/no) [no]:
```

```
>
```

```
updated: src/Entity/Post.php
```

```
Add another property? Enter the property name (or press <return> to stop adding fields):
```

```
>
```

Add another property? Enter the property name (or press <return> to stop adding fields):

> likes

Field type (enter ? to see all types) [string]:

>

Field length [255]:

>

Can this field be null in the database (nullable) (yes/no) [no]:

>

updated: src/Entity/Post.php

Add another property? Enter the property name (or press <return> to stop adding fields):

> fecha

Field type (enter ? to see all types) [string]:

> date

Can this field be null in the database (nullable) (yes/no) [no]:

> n

updated: src/Entity/Post.php



Add another property? Enter the property name (or press <return> to stop adding fields):

> contenido

Field type (enter ? to see all types) [string]:

>

Field length [255]:

>

Can this field be null in the database (nullable) (yes/no) [no]:

>

updated: src/Entity/Post.php

Add another property? Enter the property name (or press <return> to stop adding fields):

>

Success!

Next: When you're ready, create a migration with `php bin/console make:migration`

C:\wamp64\www\miAppSymfony>

# CREANDO LAS ENTIDADES

Vamos a ver el archivo que ha creado en la ruta que me ha indicado al principio de su creación, en **src/entity/Posts.php**.

Es una clase con los atributos que nosotros hemos puesto en a línea de comandos como campos de mi tabla, debemos observar que sobre cada una de esas propiedades hay una especie de comentarios que realmente no lo son, es una sintaxis especial que necesita symfony para definir el tipo de campo, tipo longitud, etc.

```

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: PostRepository::class)]
class Post
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private ?string $titulo = null;

    #[ORM\Column(length: 255)]
    private ?string $likes = null;

    #[ORM\Column(type: Types::DATE_MUTABLE)]
    private ?\DateTimeInterface $fecha = null;

    #[ORM\Column(length: 255)]
    private ?string $contenido = null;

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getTitulo(): ?string
    {
        return $this->titulo;
    }

    public function setTitulo(string $titulo): static

```

```

    public function setTitulo(string $titulo): static
    {
        $this->titulo = $titulo;

        return $this;
    }

    public function getLikes(): ?string
    {
        return $this->likes;
    }

    public function setLikes(string $likes): static
    {
        $this->likes = $likes;

        return $this;
    }

    public function getFecha(): ?\DateTimeInterface
    {
        return $this->fecha;
    }

    public function setFecha(\DateTimeInterface $fecha): static
    {
        $this->fecha = $fecha;

        return $this;
    }

    public function getContenido(): ?string
    {

```

# CREAR EL ESQUEMA, Y ACTUALIZAR LA BBDD

Una vez **terminadas todas las entidades** se debe ejecutar.

**php bin/console doctrine:schema:create**

Con este comando pasamos las entidades que hemos creado a tablas físicas, y quedan vinculadas.

A partir de ahora, cada vez que queramos actualizar las tablas, tendremos que ejecutar :

**php bin/console doctrine:schema:update --dump-sql [--force]**

- Si estamos seguros de la modificación, le pondremos la opción force

# CREANDO LAS ENTIDADES / TABLAS

## EJERCICIO :

Hacer el resto de tablas **excepto la tabla user** que se explicará después.

*Recuerda que cuando tengas creadas las clases debes crear el esquema para que pasen a ser tablas físicas en la BBDD, y comprueba que aparecen en phmyadmin.*

```
C:\wamp64\www\miAppSymfony2>php bin/console make:entity
```

```
Class name of the entity to create or update (e.g. OrangeKangaroo):
```

```
> Profession
```

```
Add the ability to broadcast entity updates using Symfony UX Turbo? (yes/no) [no]:
```

```
>
```

```
created: src/Entity/Profesion.php
```

```
created: src/Repository/ProfesionRepository.php
```

```
Entity generated! Now let's add some fields!
```

```
You can always add more fields later manually or by re-running this command.
```

```
New property name (press <return> to stop adding fields):
```

```
> nombre
```

```
Field type (enter ? to see all types) [string]:
```

```
>
```

```
Field length [255]:
```

```
>
```

```
Can this field be null in the database (nullable) (yes/no) [no]:
```

```
>
```

```
updated: src/Entity/Profesion.php
```

```
C:\wamp64\www\miAppSymfony>php bin/console make:entity
```

```
Class name of the entity to create or update (e.g. GrumpyKangaroo):
```

```
> comentario
```

```
Add the ability to broadcast entity updates using Symfony UX Turbo? (yes/no) [no]:
```

```
> comentarios
```

```
created: src/Entity/Comentario.php
```

```
created: src/Repository/ComentarioRepository.php
```

```
Entity generated! Now let's add some fields!
```

```
You can always add more fields later manually or by re-running this command.
```

```
New property name (press <return> to stop adding fields):
```

```
> comentarios
```

```
Field type (enter ? to see all types) [string]:
```

```
>
```

```
Field length [255]:
```

```
>
```

```
Can this field be null in the database (nullable) (yes/no) [no]:
```

```
>
```

```
updated: src/Entity/Comentario.php
```

```
Add another property? Enter the property name (or press <return> to stop adding fields):
```

```
> fecha
```

```
Field type (enter ? to see all types) [string]:
```

```
> ?
```

# CREACIÓN DEL ESQUEMA DE LA BBDD

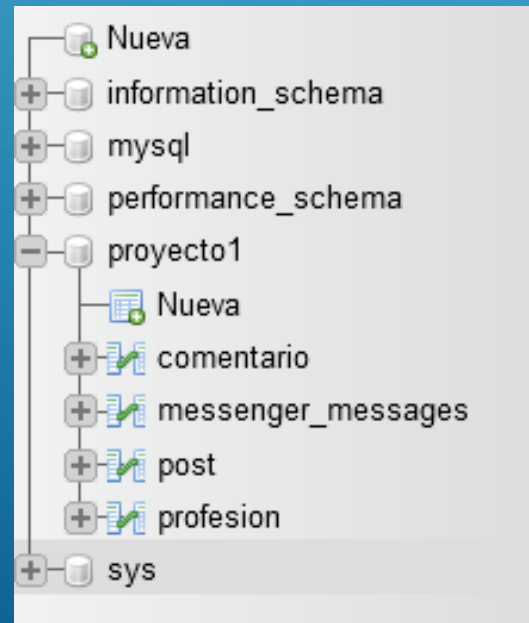
```
C:\wamp64\www\miAppSymfony>php bin/console doctrine:schema:create
```

```
!  
! [CAUTION] This operation should not be executed in a production environment!  
!
```

```
Creating database schema...
```

```
[OK] Database schema created successfully!
```

Ahora vemos que se han creado las tablas físicas en la bbdd proyecto1:





# CREACIÓN DE USUARIO

La tabla user es especial porque a través de ella van a acceder los usuarios a la aplicación. Symfony tiene un bundle que nos ayuda con el acceso seguro y su gestión, y es el que vamos a utilizar.

La instrucción es muy parecida, cambia entity por user, para gestionar esta entidad con unas medidas de seguridad:

➤ **php bin/console make:user**

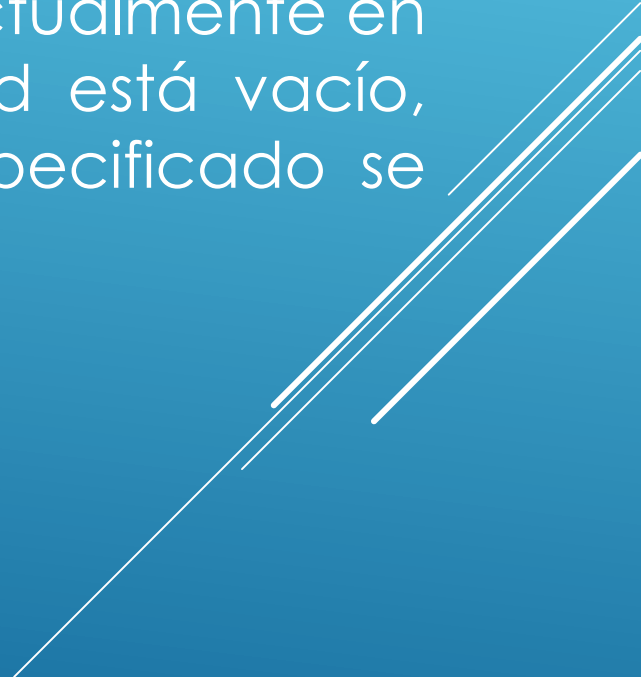
✓ Nos va a preguntar:

- Por su nombre que en nuestra bbdd será también User, damos que sí
- Si queremos gestionarlo a través de doctrine, decimos que si
- Con que campo quiero entrar, le indico que con el email
- Me pregunta si quiero encriptar las contraseñas, le indico que si

Y con esto termino la entidad, y ahora vamos al fichero generado en src/entity

# CREACIÓN DE USUARIO

Esta entidad hará cambios en el fichero security.yaml, actualmente en la sección de los proveedores de usuarios de la bdd está vacío, cuando generemos la entidad User con el bundle especificado se configurará automáticamente.

Three white lines of varying lengths and slopes are positioned in the bottom right corner of the slide, creating a modern, abstract graphic element.

```

security:
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#the-firewall

            # https://symfony.com/doc/current/security/impersonating\_user.html
            # switch_user: true

    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }

when@test:
    security:
        password_hashers:
            # By default, password hashers are resource intensive and take time. This is
            # important to generate secure password hashes. In tests however, secure hashes
            # are not important, waste resources and increase test times. The following
            # reduces the work factor to the lowest possible values.
            Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
                algorithm: auto
                cost: 4 # Lowest possible value for bcrypt
                time_cost: 3 # Lowest possible value for argon
                memory_cost: 10 # Lowest possible value for argon

```

```

security:
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#the-firewall

            # https://symfony.com/doc/current/security/impersonating\_user.html
            # switch_user: true

    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }

when@test:
    security:
        password_hashers:
            # By default, password hashers are resource intensive and take time. This is
            # important to generate secure password hashes. In tests however, secure hashes
            # are not important, waste resources and increase test times. The following
            # reduces the work factor to the lowest possible values.
            Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
                algorithm: auto
                cost: 4 # Lowest possible value for bcrypt

```

# CREACIÓN DE USUARIO

```
C:\wamp64\www\miAppSymfony>php bin/console make:user
```

```
The name of the security user class (e.g. User) [User]:
```

```
>
```

```
Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
```

```
>
```

```
Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
```

```
> email
```

```
Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed  
server).
```

```
Does this app need to hash/check user passwords? (yes/no) [yes]:
```

```
>
```

```
created: src/Entity/User.php
```

```
created: src/Repository/UserRepository.php
```

```
updated: src/Entity/User.php
```

```
updated: config/packages/security.yaml
```

Success!

Next Steps:

- Review your new `App\Entity\User` class.
- Use `make:entity` to add more fields to your `User` entity and then run `make:migration`.
- Create a way to authenticate! See <https://symfony.com/doc/current/security.html>

# CREACIÓN DE USUARIO

Observando este archivo vemos que **faltan dos campos** de la entidad(baneado, y nombre) que yo quiero tener y no me ha dejado introducir, para poder hacerlo se puede introducir las dos propiedades volviendo a editar la entidad :

➤**php bin/console make:entity,**

Al dar el nombre nos dirá que está ya creada, y nos pregunta si queremos añadir mas campos, en este punto seguimos haciendo lo mismo que cuando la creamos por primera vez, y se añaden en la entidad tanto los campos como los métodos set y get de esos campos, comprobamos que es así:

```

namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;

#[ORM\Entity(repositoryClass: UserRepository::class)]
#[ORM\UniqueConstraint(name: 'UNIQ_IDENTIFIER_EMAIL', fields: ['email'])]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 180)]
    private ?string $email = null;

    /**
     * @var list<string> The user roles
     */
    #[ORM\Column]
    private array $roles = [];

```

```

    /**
     * @var string The hashed password
     */
    #[ORM\Column]
    private ?string $password = null;

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getEmail(): ?string
    {
        return $this->email;
    }

    public function setEmail(string $email): static
    {
        $this->email = $email;

        return $this;
    }

    /**
     * A visual identifier that represents this user.
     *
     * @see UserInterface
     */
    public function getUserIdentifier(): string
    {
        return (string) $this->email;
    }

```

```
C:\wamp64\www\miAppSymfony2>php bin/console make:entity

Class name of the entity to create or update (e.g. DeliciousChef):
> User

Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):
> baneado

Field type (enter ? to see all types) [string]:
> boolean

Can this field be null in the database (nullable) (yes/no) [no]:
> yes

Add another property? Enter the property name (or press <return>):
> nombre

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/User.php

Add another property? Enter the property name (or press <return>):
>

Success!
```



```

namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;

#[ORM\Entity(repositoryClass: UserRepository::class)]
#[ORM\UniqueConstraint(name: 'UNIQ_IDENTIFIER_EMAIL', fields: ['email'])]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 180)]
    private ?string $email = null;

    /**
     * @var list<string> The user roles
     */
    #[ORM\Column]
    private array $roles = [];

    /**
     * @var string The hashed password
     */
    #[ORM\Column]
    private ?string $password = null;

```

```

#[ORM\Column]
private ?bool $banned = null;

#[ORM\Column(length: 255)]
private ?string $nombre = null;

public function getId(): ?int
{
    return $this->id;
}

public function getEmail(): ?string
{
    return $this->email;
}

public function setEmail(string $email): static
{
    $this->email = $email;

    return $this;
}

/**
 * A visual identifier that represents this user.
 *
 * @see UserInterface
 */
public function getUserIdentifier(): string

```



# ACTUALIZACIÓN DEL ESQUEMA

Ahora actualizamos el esquema para que aparezca la tabla user dentro de nuestra BBDD física.

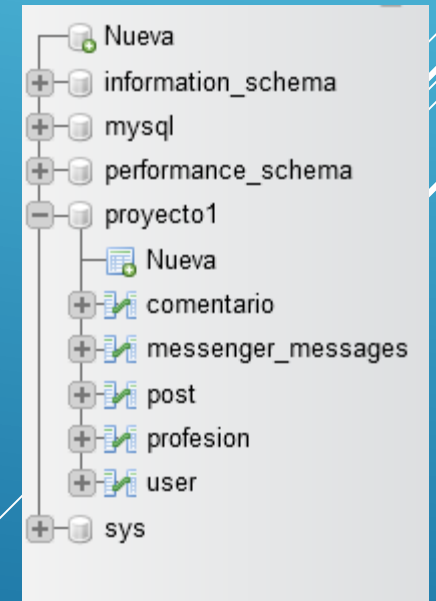
Como hemos dicho anteriormente, a partir de ahora, cada vez que queramos actualizar las tablas, tendremos que ejecutar esto en el entorno de desarrollo:






- **php bin/console doctrine:schema:update --dump-sql [--force]**
- Si estamos seguros de la modificación, le pondremos la opción **forcé**

```
C:\wamp64\www\miAppSymfony>php bin/console doctrine:schema:update --force
Updating database schema...
```










```
1 query was executed
```

```
[OK] Database schema updated successfully!
```









 	proyecto1 comentario
	id : int
	comentarios : varchar(255)
	fecha : date

 	proyecto1 profesion
	id : int
	nombre : varchar(255)

 	proyecto1 messenger_messages
	id : bigint
	body : longtext
	headers : longtext
	queue_name : varchar(190)
	created_at : datetime
	available_at : datetime
	delivered_at : datetime

 	proyecto1 post
	id : int
	titulo : varchar(255)
	likes : varchar(255)
	fecha : date
	contenido : varchar(255)

 	proyecto1 user
	id : int
	email : varchar(180)
	roles : json
	password : varchar(255)

# MODELO ENTIDAD RELACIÓN ACTUAL

Como podemos observar el modelo entidad relación solo tiene las entidades, le faltan las relaciones para poder comunicarse entre ellas.


Además vemos que hay una tabla que nosotros no hemos creado, es la tabla `messenger_messages` que symfony crea automáticamente y permite enviar y procesar mensajes de forma asíncrona, lo que mejora la respuesta de la aplicación al usuario y evita bloqueos innecesarios.

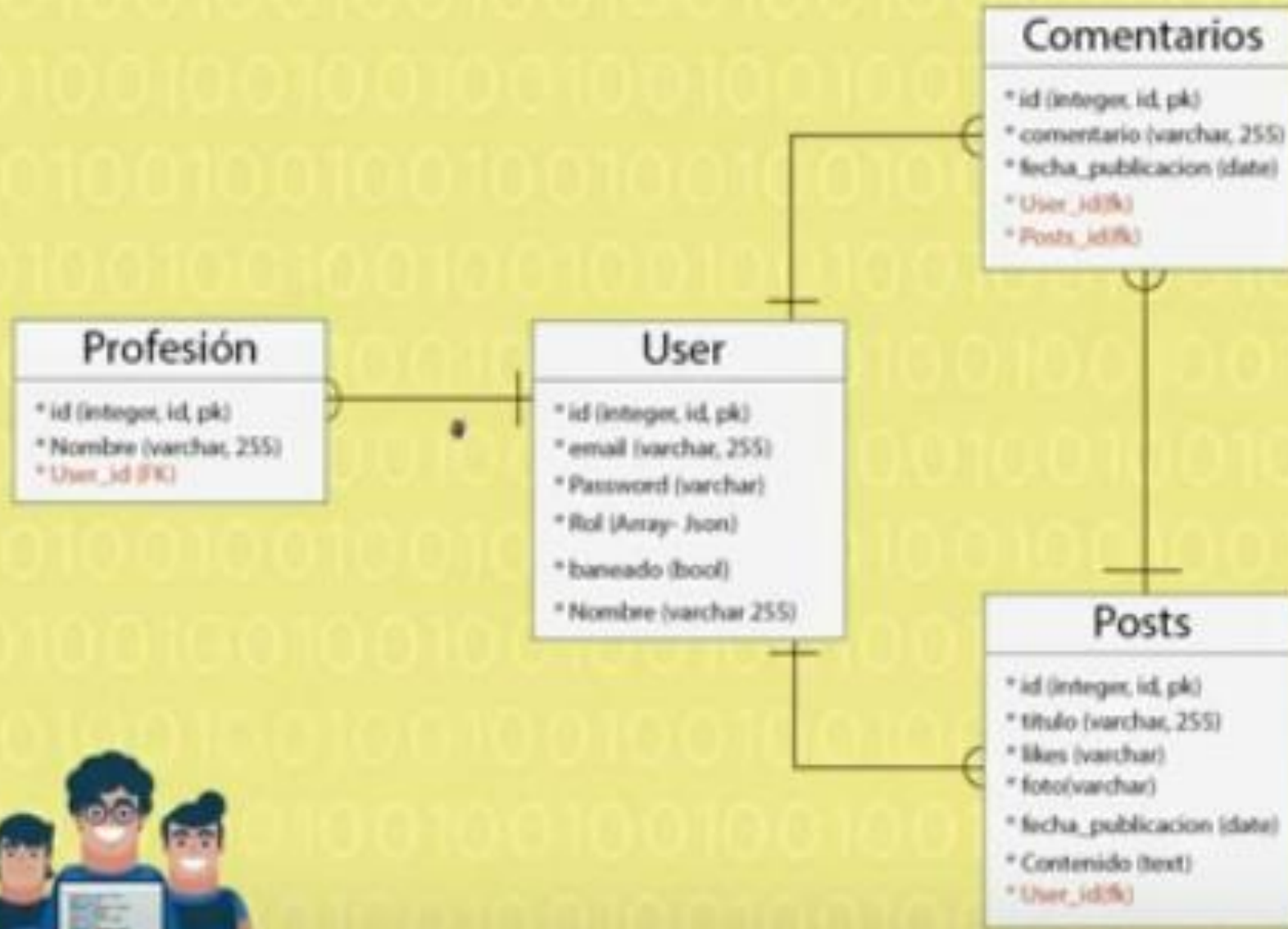
La tabla sirve para guardar los mensajes que se quedan en cola para su posterior envío

# CREANDO RELACIONES

En estos momentos no tenemos ninguna tabla relacionada, son unas independientes de las otras.

Para reflejar estos conceptos symfony tiene su propia sintaxis, en la que define toda la información en esa especie de comentarios, pero lo hará automáticamente editando de nuevo las entidades en las que queramos añadir las claves foráneas.

Several white lines of varying lengths and orientations are positioned in the bottom right corner of the slide, creating a modern, abstract graphic element.



# CREANDO RELACIONES (COMENTARIO-POST->N:1)

Una vez editada la entidad nos indicará que esta entidad ya existe, y nos preguntará los siguientes conceptos:

- ✓ Si queremos añadir algún campo, aquí ponemos el nombre de la propiedad que será FK.
- ✓ El tipo de campo que es, le decimos que es relacional(**relation**).
- ✓ El nombre de la clase con la que se relaciona(**nombre de la tabla**)
- ✓ El tipo de relación que une esas dos entidades: en este caso es de many to one porque un comentario solo pertenece a 1 post pero un post puede tener muchos comentarios.
- ✓ Si ese campo puede ser o no nulo.
- ✓ Si quiero que añada los métodos para poder acceder a la información del usuario que hace los comentarios, decimos que sí para que genere los métodos automáticamente.
- ✓ Si quiero que se haga el borrado en cascada, digo que sí para que genere los métodos automáticamente.

```
C:\wamp64\www\miAppSymfony2>php bin/console make:entity
```

```
Class name of the entity to create or update (e.g. GentleKangaroo):
```

```
> Comentario
```

```
Your entity already exists! So let's add some new fields!
```

```
New property name (press <return> to stop adding fields):
```

```
> post
```

```
Field type (enter ? to see all types) [string]:
```

```
> relation
```

```
What class should this entity be related to?:
```

```
> Post
```

```
What type of relationship is this?
```

Type	Description
ManyToOne	Each <b>Comentario</b> relates to (has) <b>one</b> <b>Post</b> . Each <b>Post</b> can relate to (can have) <b>many</b> <b>Comentario</b> objects.
OneToMany	Each <b>Comentario</b> can relate to (can have) <b>many</b> <b>Post</b> objects. Each <b>Post</b> relates to (has) <b>one</b> <b>Comentario</b> .
ManyToMany	Each <b>Comentario</b> can relate to (can have) <b>many</b> <b>Post</b> objects. Each <b>Post</b> can also relate to (can also have) <b>many</b> <b>Comentario</b>
OneToOne	Each <b>Comentario</b> relates to (has) exactly <b>one</b> <b>Post</b> . Each <b>Post</b> also relates to (has) exactly <b>one</b> <b>Comentario</b> .

```
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
```

```
> ManyToOne
```

Is the Comentario.posts property allowed to be null (nullable)? (yes/no) [yes]:

>

Do you want to add a new property to post so that you can access/update Comentario objects from it - e.g. \$post->getComentarios()? (yes/no) [yes]:

>

A new property will also be added to the post class so that you can access the related Comentario objects from it.

New field name inside post [comentarios]:

>

updated: src/Entity/Comentario.php

updated: src/Entity/Post.php

Add another property? Enter the property name (or press <return> to stop adding fields):

>

Success!

Next: When you're ready, create a migration with `php bin/console make:migration`



# MODIFICAR ENTIDADES

Una vez hecha las relaciones, podemos también modificar cosas de nuestras entidades, por ejemplo **quiero forzar a que algunos campos sean únicos en la base de datos** como el título de un post. Para ello lo único que tenemos que hacer, es añadir a la **anotación** de la columna el parámetro `unique: true`. Y en `baneado`, y roles de `user.php` que puedan ser nulos:

```
#[ORM\Entity(repositoryClass: PostRepository::class)]
class Post
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private ?string $titulo = null;
```



```
#[ORM\Id]
#[ORM\GeneratedValue]
#[ORM\Column]
private ?int $id = null;

#[ORM\Column(length: 255, unique: true)]
private ?string $titulo = null;
```

```
/**
 * @var string The hashed password
 */
#[ORM\Column]
private ?string $password = null;

#[ORM\Column]
private ?bool $baneado = null;
```



```
#[ORM\Column(nullable: true)]
private array $roles = [];

/**
 * @var string The hashed password
 */
#[ORM\Column]
private ?string $password = null;

#[ORM\Column(nullable: true)]
private ?bool $baneado = null;
```

# MODIFICAR ENTIDADES

Ahora ya tenemos las entidades listas, están hechas las relaciones y se han modificado las columnas baneado y título, así que el siguiente paso es actualizar de nuevo mi esquema:

➤ **php bin/console doctrine:schema:update --force**

```
C:\wamp64\www\miAppSymfony>php bin/console doctrine:schema:update --force
Updating database schema...
```

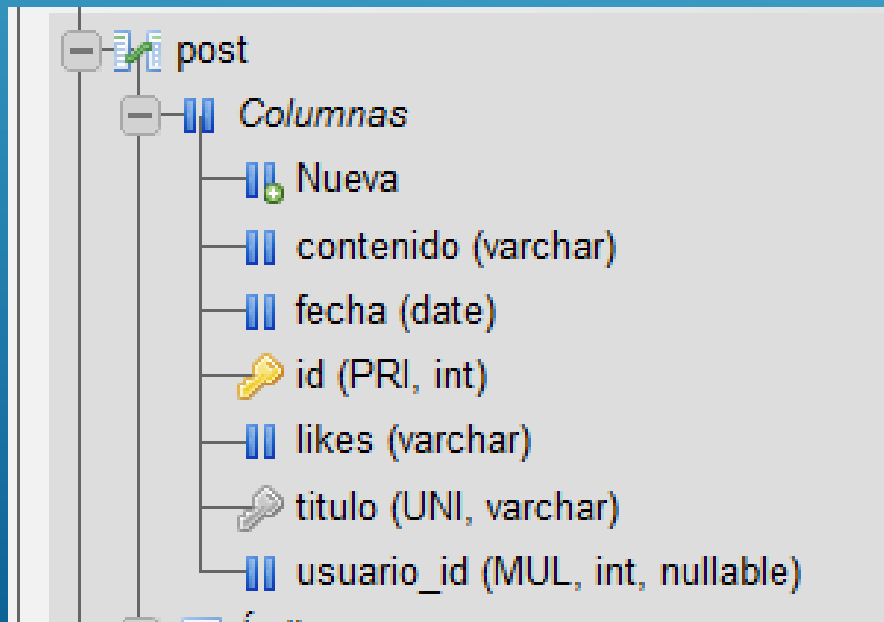
```
    12 queries were executed
```

```
[OK] Database schema updated successfully!
```

# MODIFICAR ENTIDADES

Ahora vamos a verificar el ME/R, así como la estructura y la vista de relaciones de la tabla comentario, por ejemplo.

Y en la tabla post verifico que el titulo lo ha cambiado a único:



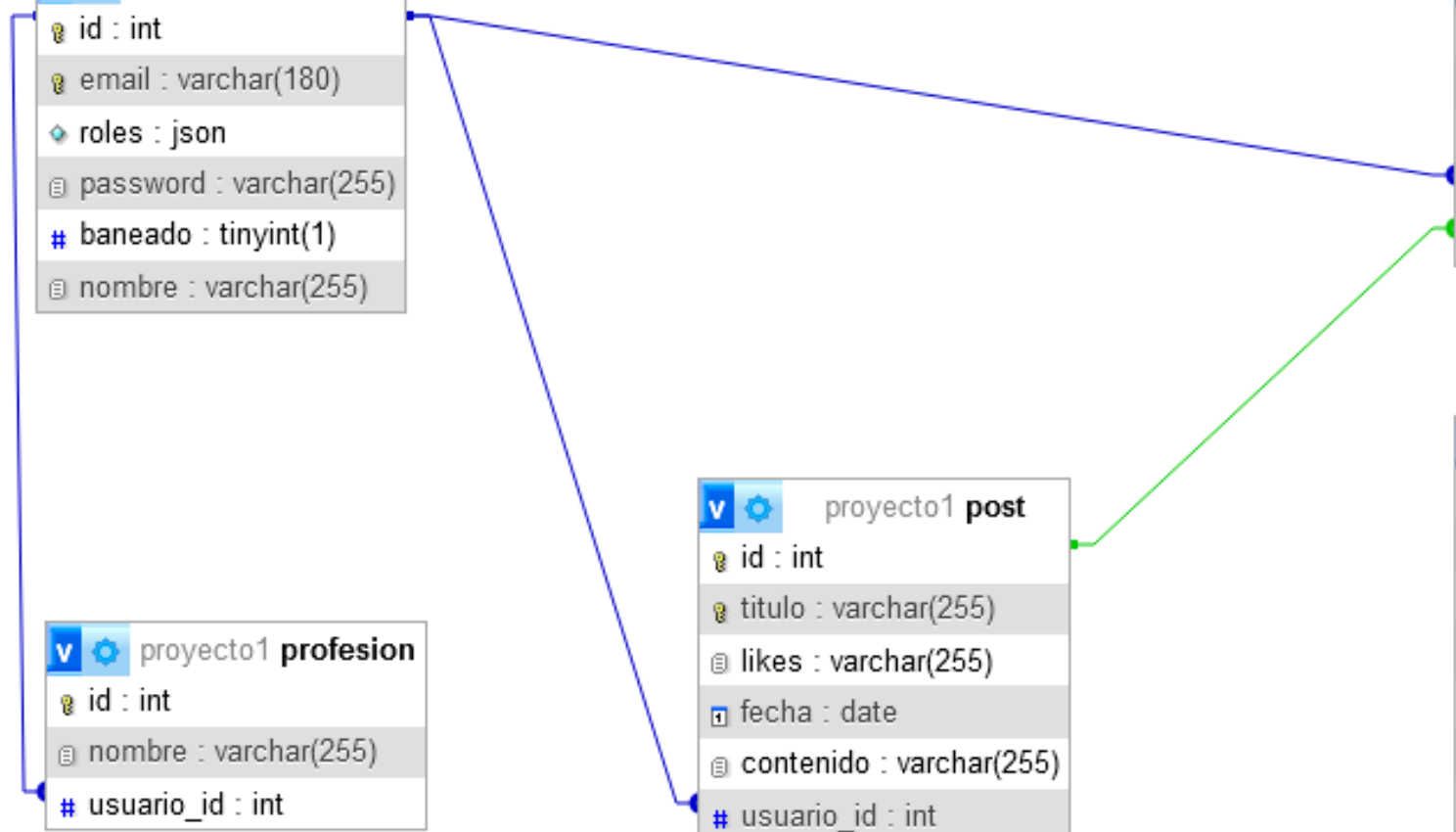
projecto1 user
id : int
email : varchar(180)
roles : json
password : varchar(255)
# baneado : tinyint(1)
nombre : varchar(255)

projecto1 profesion
id : int
nombre : varchar(255)
# usuario_id : int

projecto1 post
id : int
titulo : varchar(255)
likes : varchar(255)
fecha : date
contenido : varchar(255)
# usuario_id : int

projecto1 comentario
id : int
comentarios : varchar(255)
fecha : date
# usuario_id : int
# posts_id : int

projecto1 messenger_messages
id : bigint
body : longtext
headers : longtext
queue_name : varchar(190)
created_at : datetime
available_at : datetime
delivered_at : datetime





Estructura de tabla



Vista de relaciones

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
<input type="checkbox"/>	1	id	int			No	Ninguna		AUTO_INCREMENT	Cambiar  Eliminar Más
<input type="checkbox"/>	2	comentarios	varchar(255)	utf8mb4_unicode_ci		No	Ninguna			Cambiar  Eliminar Más
<input type="checkbox"/>	3	fecha	date			No	Ninguna			Cambiar  Eliminar Más
<input type="checkbox"/>	4	usuario_id	int			Sí	NULL			Cambiar  Eliminar Más
<input type="checkbox"/>	5	posts_id	int			Sí	NULL			Cambiar  Eliminar Más



Estructura de tabla



Vista de relaciones

## Restricciones de clave foránea

Acciones	Propiedades de la restricción		Columna	Restricción de clave foránea (INNODB)		
				Base de datos	Tabla	Columna
Eliminar	ON DELETE	RESTRICT	posts_id	proyecto1	post	id
	ON UPDATE	RESTRICT	+ Añadir columna			
Eliminar	ON DELETE	RESTRICT	usuario_id	proyecto1	user	id
	ON UPDATE	RESTRICT	+ Añadir columna			

# COMANDOS DE INTERÉS

1. `php bin/console cache:pool:clear cache.global_clearer:` Limpiar cache
2. `php bin/console:` Nos lista todos los comandos que tenemos hasta ahora mismo.
3. `php bin/console list make:`  
Listarlos todos los comandos que permiten generar las ayudas

# COMANDOS DE INTERÉS

- Creación de la BBDD.
  - `php bin\console doctrine:database:create`
- Borrado de la BBDD.
  - `php bin\console doctrine:database:drop`
- Creación de la Entidad
  - `php bin\console doctrine:make:entity`

# COMANDOS DE INTERÉS

- Creación de tablas físicas
  - `php bin/console doctrine:schema:create`
- Borrado solo las tablas:
  - `php bin\console doctrine:schema:drop --force`
- Modificación de tablas:
  - `php bin/console doctrine:schema:update --force`



# PASOS PARA CREAR LA BBDD

1. Configuro al bbdd que viene por defecto en el fichero.env

2. Creamos la base de datos:

- `php bin/console doctrine:database:create`

3. Creamos las entidades y relaciones:

- `php bin/console doctrine:make:entity`

En este punto están creadas Entidades y la BBDD por tanto a partir de las entidades genero las tablas en la bbdd.

4. Con este comando se sincronizará nuestra aplicación con la bbdd

- `php bin\console doctrine:schema:create`

Y ya podemos comprobar que están las tablas creadas

# PRÁCTICA

Genera tú todas las relaciones de las tablas, y actualiza el esquema de la BBDD, siguiendo las instrucciones de las diapositivas.

Several thin, white, parallel diagonal lines are positioned in the bottom right corner of the slide, extending from the right edge towards the center.