



## 10) Le pack Profiler

Le Profiler est un outil extrêmement puissant de Symfony. C'est un outil de développement qui va nous donner des informations détaillées sur ce qui s'est passé lors de l'exécution d'une requête. Il faut bien configurer le profiler en mode développement et jamais en mode production.



Pour installer le composer profiler-pack ( nous grâce au `–full` on l'a déjà) :

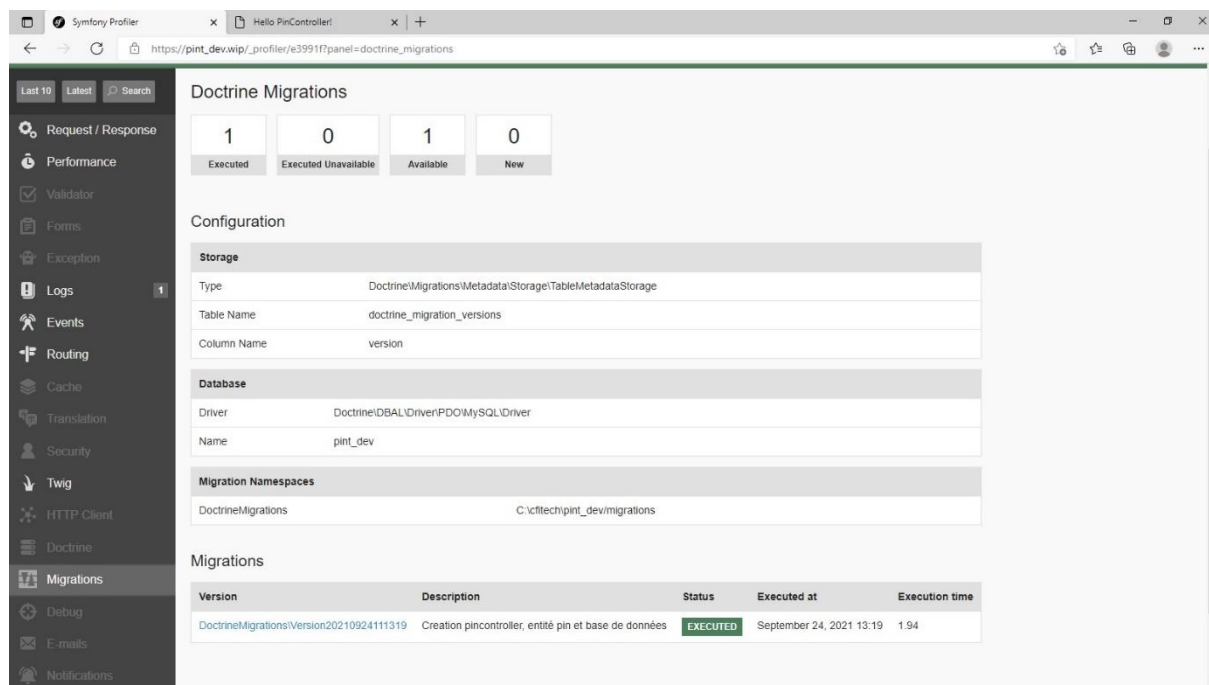
- `composer req profiler –dev` (installe profiler symfony/web debug toolbar symfony)
- `composer require debug –dev` (va installer tout ce qui nous permet de débbuger)

Ils permettent d'avoir le `web_profiler` toolbar, c'est une barre ou on peut avoir plusieurs informations sur la page et ça va nous permettre de pouvoir débbuger.

Si ça ne marche pas il faut nettoyer le cache :

- `symfony console cache:clear`

Lorsque l'on clique sur le profiler, on a un tas d'information, si vous cliquer sur « **migrations** » vous pourrez voir qu'il a vu qu'on a fait une migration et nous donne même la description que l'on avait mis dans le fichier de migrations.



**Doctrine** c'est là qu'on verra quand on fera des requêtes vers la base de données. Pour l'instant nous n'avons rien car on n'a pas encore fait de requêtes.

**Routing** nous permet de voir nos routes. Normalement vous avez dû appeler votre route « **app\_home** » avec comme path (chemin) « **/** ».

## **11) Ajout d'attributs à une « entity » existante**

Nous allons rajouter une date de création d'un pin comme attribut qui ne changera jamais une fois crée (« **createdAt** »), ainsi qu'un attribut « **updatedAt** » qui une fois qu'on modifiera un pin, il mettra à jours la date de modification automatiquement.

Pour rajouter des attributs à une entité déjà existante, il suffit de taper la commande dans la console

- **symfony console make:entity**

Il demande le nom de l'entité, on remet « Pin » (il va voir qu'il existe déjà donc il va demander si on veut rajouter de nouveaux champs)

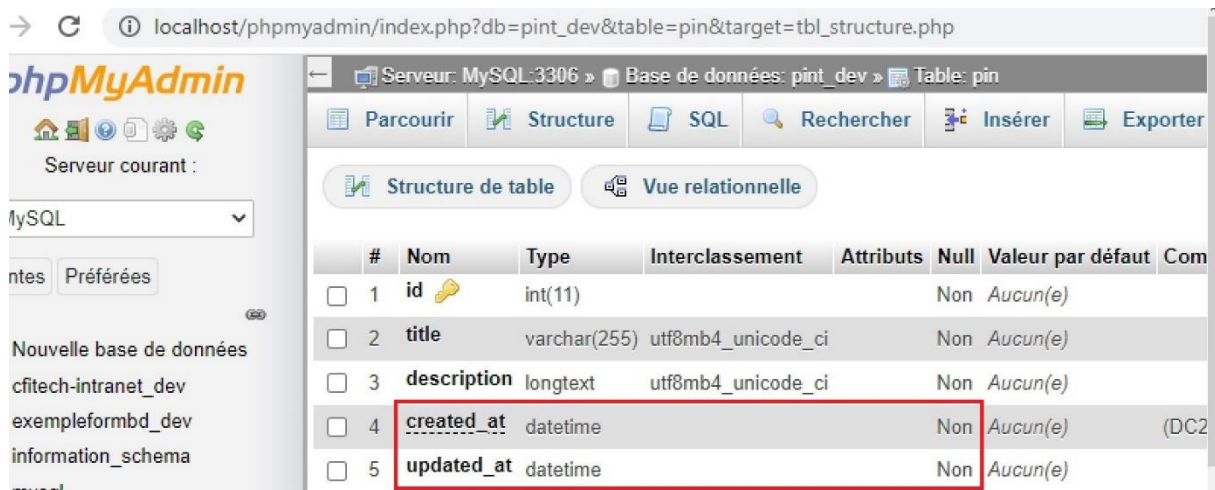
- **createdAt, datetime, champs no null**
- **updatedAt, datetime, champs no null**

Appuyer ensuite encore sur « Enter » pour quitter.

## Exo(10-15 min)

Vous verrez qu'il vous proposera de faire directement une migration. Nous avons vu la commande dans le PDF « Symfony part 1 », je vous laisse faire la migration vers la base de données en mettant comme description : « **Ajout de createdAt et updatedAt dans Pin** ».

Vérifier que les nouveaux champs createdAt et updatedAt ont bien été envoyé dans la base de données.



localhost/phpmyadmin/index.php?db=pint\_dev&table=pin&target=tbl\_structure.php

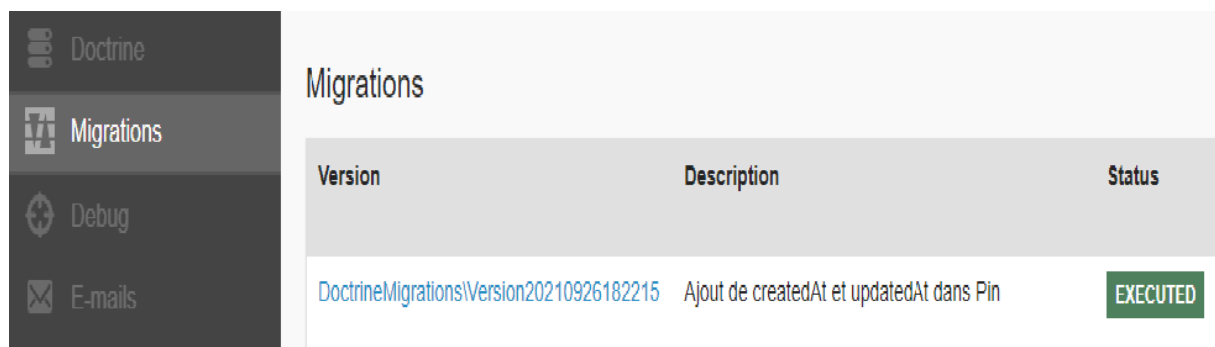
phpMyAdmin

Serveur courant : MySQL

Structure de table

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Com
<input type="checkbox"/>	1 id	int(11)			Non	Aucun(e)	
<input type="checkbox"/>	2 title	varchar(255)	utf8mb4_unicode_ci		Non	Aucun(e)	
<input type="checkbox"/>	3 description	longtext	utf8mb4_unicode_ci		Non	Aucun(e)	
<input type="checkbox"/>	4 <u>created_at</u>	datetime			Non	Aucun(e)	(DC2
<input type="checkbox"/>	5 updated_at	datetime			Non	Aucun(e)	

Ensuite vérifier dans le Profiler si on voit bien le message de description dans l'onglet migrations.



Doctrine

Migrations

Debug

E-mails

### Migrations

Version	Description	Status
Doctrine\Migrations\Version20210926182215	Ajout de createdAt et updatedAt dans Pin	EXECUTED

Je vous demanderai aussi de faire un commit avec une description de ce que vous avez fait sur GitHub que ce soit via Visual studio ou en ligne de commande.

## 12) Configuration de createdAt et updatedAt

Nous voulons faire en sorte que quand on créera un pin, la date actuelle sera envoyée automatiquement en base de données et que quand on modifiera un pin, il montrera la date de modification.

Dans la classe « **Pin** » qui se situe dans le dossier **src/Entity** on va ajouter (voir photo ci-dessous) :

```
- /**
    * @ORM\Entity(repositoryClass=PinRepository::class)
    * @ORM\Table(name="pin")
    * @ORM\HasLifecycleCallbacks
    */
```

(PHP 8 : #[ORM\Table(name="pins")])

#[ORM\HasLifecycleCallbacks])

« **HasLifecycleCallbacks** » c'est pour pouvoir utiliser prePersist et preUpdate, on va écouter les différents événements du cycle de vie.

On va mettre ces deux nouveaux attributs dans PIN avec un « Optional » qui va donner l'heure actuelle par défaut :

```
8  /**
9  * @ORM\Entity(repositoryClass=PinRepository::class)
10 * @ORM\Table(name="pin")
11 * @ORM\HasLifecycleCallbacks
12 */
13 class Pin
14 {
15     /**
16     * @ORM\Id
17     * @ORM\GeneratedValue
18     * @ORM\Column(type="integer")
19     */
20     private $id;
21
22     /**
23     * @ORM\Column(type="string", length=255)
24     */
25     private $title;
26
27     /**
28     * @ORM\Column(type="text")
29     */
30     private $description;
31
32     /**
33     * @ORM\Column(type="datetime", options={"default": "CURRENT_TIMESTAMP"})
34     */
35     private $createdAt;
36
37     /**
38     * @ORM\Column(type="datetime", options={"default": "CURRENT_TIMESTAMP"})
39     */
40     private $updatedAt;
```

```

- /**
    * @ORM\Column(type="datetime", options={"default": "CURRENT_TIMESTAMP"})
 */
    private $createdAt;

/**
    * @ORM\Column(type="datetime", options={"default": "CURRENT_TIMESTAMP"})
 */
    private $updatedAt;

```

(PHP 8 : #[ORM\Column(type="datetime", options:['default'=>'CURRENT\_TIMESTAMP'])])

On va ensuite créer une méthode qui pourrait avoir n'importe quel nom, mais généralement les `updatedAt` et `createdAt` sont des timestamps (des événements liés au temps) :

```

95  /**
96  * @ORM\PrePersist //cette methode on veut l'appeler avant un persist (avant creation d'un pin on appel
97  * @ORM\PreUpdate //cette methode on veut l'appeler avant un update (avant la modification d'un pin)
98  */
99  public function updateTimestamps(){
100      if ($this->getCreatedAt()=== null){
101          $this->setCreatedAt(new \DateTimeImmutable); //DateTimeImmutable, cest qu'on ne peut pas modifier
102      }
103      $this->setUpdatedAt(new \DateTimeImmutable);
104  }
105

```

```

- /**
    * @ORM\PrePersist
    * @ORM\PreUpdate
 */
    public function updateTimestamps(){
        if ($this->getCreatedAt()=== null){
            $this->setCreatedAt(new \DateTimeImmutable);
        }
        $this->setUpdatedAt(new \DateTimeImmutable);
    }

```

On peut stopper le serveur et le relancer.

## 13) Persister nos données avec l'EntityManager & Repository

Un **Repository** Doctrine est un objet dont la responsabilité est de récupérer une collection d'objets. Les repositories ont accès à deux objets principalement :

- EntityManager : Permet de manipuler nos entités ;
- QueryBuilder : Un constructeur de requêtes qui permet de créer des requêtes personnalisé.

Chaque entité dispose d'un repository par défaut accessible dans tous les contrôleurs de nos applications. D'ailleurs vous pouvez voir qu'on a un dossier repository avec PinRepository.

L'**EntityManager** est l'objet permettant d'effectuer les opérations liées à l'altération des données, à savoir les requêtes de type **INSERT**, **UPDATE** et **DELETE**. Au niveau de votre application, même si vous manipulez des entités, qui correspondent aux données présentes en base, il ne suffit pas de modifier la valeur d'une propriété pour que Doctrine fasse la synchronisation en base de données. Vous devez gérer ces opérations avec l'EntityManager.

Afin de persister notre pin dans la Base de Données, on va utiliser l'ENTITYMANAGER, qui va nous permettre de gérer nos entités, allez dans « PinController » et modifiez le :

```
1  <?php
2
3  namespace App\Controller;
4  use App\Entity\Pin;
5  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6  use Symfony\Component\HttpFoundation\Response;
7  use Symfony\Component\Routing\Annotation\Route;
8  use App\Repository\PinRepository;
9
10
11
12 class PinController extends AbstractController
13 {
14     /**
15      * @Route("/", name="app_home")
16      */
17     public function index(PinRepository $repo): Response
18     {
19         $pin = new Pin;
20         $pin->setTitle('Pin 1');
21         $pin->setDescription('Description Pin 1');
22
23         $em = $this->getDoctrine()->getManager();
24         $em->persist($pin);
25         $em->flush();
26
27         return $this->render('pin/index.html.twig', ['pins' => $repo->findAll()]);
28     }
29 }
30 }
```

- **\$this->getDoctrine()->getManager();**

Le \$this est une référence à un objet PHP. La méthode getDoctrine() elle existe parce qu'on hérite de AbstractController, et via doctrine on peut accéder à entityManager en faisant getManager(), cette ligne de code va donc nous donner un entityManager.

- `$em->persist($pin);`

Il va permettre d'informer notre EntityManager qu'on veut persister notre objet.

- `$em->flush();`

Le flush c'est comme si on disait go pour persister les données en Base de Données, il exécute la requête.

- `$this->render('pin/index.html.twig', ['pins' => $repo->findAll()]);`

Le render est la méthode de rendu/d'affichage des templates twig. C'est ce qu'utilise symfony pour l'affichage. Ce qui suit, c'est donc une méthode de PinRepository, le findAll qui en gros veut dire trouve les tous, donc ici il va aller récupérer en base de données tous les pins. On est dans PinRepository donc cela concerne l'entité/la table Pin. Il va les retourner sous forme de tableau qu'on va mettre dans 'pins'.

Si vous faites un refresh sur votre page web, il y aura une erreur liée au nom de variable controller\_name qu'on a enlevé dans notre code. Mais allez dans la barre du profiler ensuite dans doctrine vous verrez qu'on a bien une requête qui a été faite. Et si vous allez dans phpMyAdmin vous verrez bien que votre premier pin a bien été créé. Maintenant il ne manque plus que de l'afficher sur une page.

**Attention**, faites qu'un seul refresh sinon il recréera à chaque fois le même pin.

**PHP 8 :**

```
use Symfony\Component\Routing\Annotation\Route;
use App\Entity\Pin;
use App\Repository\PinRepository;
use Doctrine\Persistence\ManagerRegistry;

class PinController extends AbstractController
{
    #[Route('/', name: 'app_home')]
    public function index(ManagerRegistry $doctrine, PinRepository $repo): Response
    {
        $pin = new Pin;
        $pin->setTitle("Pin 1");
        $pin->setDescription("Description Pin 1");
        $em = $doctrine->getManager();
        $em->persist($pin);
        $em->flush();
        return $this->render('pin/index.html.twig', ['pins' => $repo->findAll()]);
    }
}
```

(Exo) Vous pouvez supprimer vos pins créés dans phpMyAdmin en réinitialisant les id.

Créez 5 pins différents via le PinController : pin 1, pin 2, pin 3, pin 4 et pin 5. Avec description du Pin 1, description du Pin 2, description du Pin 3, etc. Je veux que pour l'id 1 on a pin 1, pour l'id 2 on a pin 2 etc (Une fois corrigé, mettez en commentaire).

## 14) L'affichage de nos pins sur notre page avec TWIG

Ce qu'on mettra dans le fichier « **templates/pin/index.html.twig** » pour afficher les Pins :

```
1  {% extends 'base.html.twig' %}
2  {% block title %}Pinterest{% endblock %}
3  {% block body %}
4      {% set pin_count = pins|length %}
5      {{ pin_count }}
6      {% for pin in pins %}
7          <article>
8              <h1>{{ pin.getTitle() }}</h1>
9              <p>{{ pin.description }}</p>
10
11
12          </article>
13      {% else %}
14          <p> Sorry, no pin yet </p>
15      {% endfor %}
16  {% endblock %}
```

En twig quand on met `{% ....%}` du code entre, c'est pour mettre par exemple une opération, ici dans notre cas c'est une assignation, on met dans la variable `pin_count` la taille du tableau de pins. Le `set` sert à définir la variable dans twig.

Quand on a `{{...}}` c'est un appel a une variable, un affichage ou fonction php, ou un template Twig parent (`{{ parent }}`), ici c'est pour afficher le `pin_count`, ainsi que le titre et la description.

On réutilise ensuite `{% %}` pour faire une boucle avec condition, si pas de pin.

Voici le code :

```
- {% block body %}

    {% set pin_count = pins|length %}

    {{ pin_count }}

    {% for pin in pins %}

        <article>

            <h1>{{ pin.getTitle() }}</h1>

            <p>{{ pin.description|u.truncate(10,'...',false) }}</p>

            <p>Submitted {{ pin.createdAt|ago }}</p>

        </article>

    {% else %}

        <p> Sorry, no pin yet </p>

    {% endfor %}

{% endblock %}
```



Pour tronquer une description trop longue, on peut le faire en allant dans la zone où l'on affiche la description, donc dans index.html.twig en ajoutant u.truncate(taille de caractère avant de tronquer, on peut mettre ensuite une option, genre '...' qui se mettra à la fin du troncages, on peut ensuite mettre un booléen pour compter ou pas les '...' :

```
- ...  
  
    <h1>{{ pin.getTitle() }}</h1>  
  
    <p>{{ pin.description|u.truncate(10) }}</p>  
  
    ...
```

Il se trouve dans abstractString.

Il faudra d'abord installer ce composer pour que ça marche :

- **composer require twig/string-extra**

On pourra rajouter des arguments en plus comme les 3 petits points pour donner quelques choses de plus compréhensible :

- `<p>{{ pin.description|u.truncate(10,'...',false) }}</p>`

Le booléen indiquera si on veut compter que 10 caractères avec les '...' ou s'ils se mettent à la fin d'un mot. True est la valeur par défaut, ici on met false. Avec true il comptera les petits points.

On peut aussi mettre à quelle moment le pin a été soumis, toujours dans index.html.twig :

```
- ...  
  
    <p>{{ pin.description|u.truncate(8,'...',false) }}</p>  
  
    <p>Submitted {{ pin.createdAt|date }}</p>
```

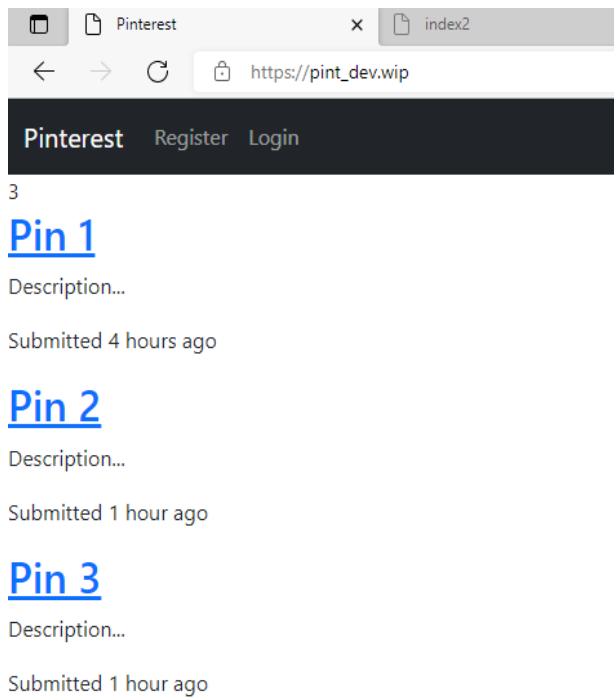
On peut montrer depuis combien de jour ça été posté, on va utiliser le filtre ago, pour se faire :

- **composer require knplabs/knp-time-bundle**

Ensuite juste remplacer

- `<p>Submitted {{ pin.createdAt|ago }}</p>`

Voici un résultat possible :



## 15) Création de la fonction show

Ici nous allons créer juste une page qui va afficher les détails d'un Pin. Quand on aura la liste des pins, dès qu'on cliquera sur le titre on sera dirigé vers les détails.

Pour pouvoir afficher les détails pour un pin, on va créer dans PinController une fonction show :

```
- //en ajoutant un requirement à coté de id, on précise qu'on attend un int.  
/**  
 * @Route("/pin/{id<[0-9]+}", name="app_pin_show", methods="GET")  
 */  
public function show(Pin $pin): Response  
{  
    return $this->render('pin/show.html.twig', compact('pin'));  
}
```

```
(PHP 8 : #[Route('/pin/{id<[0-9]+}', name:'app_pin_show', methods:'GET')]])
```

On va ensuite créer le **templates/pin/show.html.twig** :

```
{% extends 'base.html.twig' %}
{% block title %}{{ pin.title }}{% endblock %}
{% block body %}
    <article>
        <h1>{{ pin.title }}</h1>
        <p>{{ pin.description }}</p>
        <p>Submitted {{ pin.createdAt|date }}</p>
    </article>
    <a href="#">Edit</a>
    <a href="{{ path('app_home') }}">Go Back</a>
{% endblock %}
```

Il faudra pour finir dans **index.html.twig** rajouter :

```
- <h2><a href="{{ path('app_pin_show', {id: pin.id}) }}">{{ pin.title }} </a></h2>
```

A la place de `<h1>{{ pin.getTitle() }}</h1>`

Vous remarquerez qu'avec twig on peut écrire `pin.title` ou `pin.getTitle`, c'est la même. Il est intelligent donc il sait où chercher les attributs.