

Symfony





- ✦ Aller plus loin avec la couche modèle
- ✦ Réaliser des requêtes complexes

- ❖ Si on reprend la fonctionnalité n°1 du projet :
En tant qu'utilisateur, je vois les dernières offres actives sur la page d'accueil.
- ❖ Mais pour le moment la page d'accueil ne liste que les offres.
- ❖ Un emploi est considéré actif s'il a été posté il y a moins de 30 jours.
- ❖ Il va donc falloir commencer par modifier la requête effectuée dans la méthode `JobController::index`.



JobController.php :

Pour écrire une requête complexe, nous utilisons l'objet QueryBuilder fourni par Doctrine.

Cet objet nous permet de créer une requête compréhensible par l'ORM sans devoir écrire de code SQL.

L'avantage c'est que Doctrine adaptera la requête au type de base de données avec lequel il communique (SQLite, MySQL, PostgreSQL...)



```
use DateTime;
```

```
class JobController extends AbstractController
{
    /**
     * @Route("/job", name="job")
     */
    public function index(EntityManagerInterface $em):Response
    {
        //$jobs = $em->getRepository(Job::class)->findAll();
        $queryBuilder = $em->getRepository(Job::class)->createQueryBuilder('j');
        $queryBuilder->andWhere('j.createdAt > :date');
        $queryBuilder->setParameter('date', new DateTime('-30 day'));
        $jobs = $queryBuilder->getQuery()->getResult();

        return $this->render('job/index.html.twig', [
            'Listjobs' => $jobs,
        ]);
    }
}
```

Symfony

QueryBuilder

❖ Ce qui serait intéressant, ce serait d'extraire la requête du contrôleur afin qu'elle puisse être réutilisée sans devoir dupliquer le code.



- ❖ Jusqu'ici, la méthode `EntityManager::getRepository` nous permettait d'obtenir un objet générique que Doctrine utilise pour fournir des méthodes de base permettant de faire des requêtes en base de données.

- ❖ Nous allons maintenant définir une classe de type Repository. Les objets de types Repository contiennent des méthodes permettant de récupérer des données en base.



❖ On va donc créer une nouvelle classe dans un sous dossier
Repository : **JobRepository.php**

```
<?php

namespace App\Repository;

use Doctrine\ORM\EntityRepository;
use DateTime;

class JobRepository extends EntityRepository
{
    public function findActive(DateTime $date)
    {
        return $this->createQueryBuilder('j')
            ->andWhere('j.createdAt > :date')
            ->setParameter('date', $date)
            ->getQuery()
            ->getResult();
    }
}
```



- ❖ Une fois la classe créée, il va falloir modifier la configuration du mapping de l'entité Job pour indiquer à Doctrine la classe que l'ORM devra utiliser pour accéder aux données.



Symfony

QueryBuilder

Job.orm.yml

```
# config/doctrine/mapping/Job.orm.yml
App\Entity\Job:
  type: entity
  repositoryClass: App\Repository\JobRepository
...
```



Symfony

QueryBuilder

- ❖ Enfin, il faut modifier le code de notre contrôleur pour faire uniquement appel à la méthode `findActive`



JobController.php

Nous utilisons maintenant une classe pour la récupération des données de notre offre d'emploi, ce qui permet d'isoler le code dédié à la récupération des données et permet ainsi d'améliorer la maintenabilité du projet



```
<?php
```

```
namespace App\Controller;
```

```
use App\Entity\Job;
```

```
use Doctrine\ORM\EntityManagerInterface;
```

```
use Symfony\Component\HttpFoundation\Response;
```

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
```

```
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
```

```
use Symfony\Component\Routing\Annotation\Route;
```

```
use DateTime;
```

```
class JobController extends AbstractController
```

```
{
```

```
/**
```

```
 * @Route("/job", name="job")
```

```
 */
```

```
public function index(EntityManagerInterface $em):Response
```

```
{
```

```
    $jobs = $em->getRepository(Job::class)->findActive(new DateTime('-30 day'));
```

```
    return $this->render('job/index.html.twig', [
```

```
        'Listjobs' => $jobs,
```

```
    ]);
```

```
}
```

Symfony

QueryBuilder

- ❖ Il est possible de consulter la requête générée par Doctrine en consultant les logs générés par l'application.
- ❖ Par défaut, Symfony crée les logs sur la partie standard et sont donc consultables directement sur le terminal
- ❖ On peut également accéder à ces informations via la barre de debug au lancement de l'application



Symfony

QueryBuilder

The screenshot shows a web application interface. On the left, there is a Microsoft logo. To its right, the heading 'Web Designer' is followed by a paragraph of Lorem ipsum text. Below the text, it says 'Posté à 07/22/2019' and there is a blue button labeled 'En savoir plus'. A pagination control shows '« 1 »'. A dark overlay box displays database profiler statistics: 'Database Queries' (1), 'Query time' (5.91 ms), 'Invalid entities' (0), and 'Second Level Cache' (disabled). At the bottom, a status bar shows '127.0.0.1:8000/_profiler/3ba530?panel=db', '75 in 21.31 ms', 'anon.', '12 ms', and '1 in 5.91 ms'. The Symfony version '4.3.2' is in the bottom right corner.


Database Queries	1
Query time	5.91 ms
Invalid entities	0
Second Level Cache	disabled

En cliquant sur « Database Queries » nous accédons à l'ensemble des requêtes exécutées sur la page



Symfony

QueryBuilder


 **Symfony Profiler**


search on symfony.com


http://127.0.0.1:8000/job


Method: GET HTTP Status: 200 IP: 127.0.0.1 Profiled on: Wed, 24 Jul 2019 07:43:37 +0000 Token: 3ba530


Last 10 Latest


 Request / Response


 Performance


 Validator


 Forms


 Exception


 Logs


 Events


 Routing


 Cache


 Translation


 Security

 Twig

 Doctrine

 E-mails

 Debug

 Configuration

Query Metrics

1	1	5.91 ms	0
Database Queries	Different statements	Query time	Invalid entities

Queries

[Group similar statements](#)

#	Time	Info
1	5.91 ms	<pre>SELECT j0_id AS id_0, j0_type AS type_1, j0_company AS company_2, j0_logo AS logo_3, j0_url AS url_4, j0_position AS position_5, j0_location AS location_6, j0_description AS description_7, j0_how_to_apply AS how_to_apply_8, j0_token AS token_9, j0_is_public AS is_public_10, j0_is_activated AS is_activated_11, j0_email AS email_12, j0_expires_at AS expires_at_13, j0_created_at AS created_at_14, j0_updated_at AS updated_at_15, j0_category_id AS category_id_16 FROM job j0 WHERE j0_created_at > ?</pre> <p>Parameters:</p> <pre>["2019-06-24 07:43:37"]</pre> <p>View formatted query View runnable query Explain query</p>

Database Connections

Name	Service
default	doctrine.dbal.default_connection



- ❖ Pour optimiser notre code, nous allons créer une constante qui permettra de masquer le calcul de date et ainsi éviter des erreurs.
- ❖ Par ailleurs, nous allons également utiliser le champ `expiresAt` pour stocker la date d'expiration d'une offre plutôt que de devoir la calculer.



Symfony

QueryBuilder

- ❏ Tout comme pour les dates de création et de modification des offres, nous allons utiliser le gestionnaire d'événement de Doctrine pour mettre à jour la valeur du champ automatiquement



Symfony

QueryBuilder

❏ Tout d'abord, nous mettons à jour le code de notre entité **Job.php**

```
<?php

namespace App\Entity;
use App\Helper\SlugifyHelper;
use Doctrine\Common\Persistence\Event\LifecycleEventArgs;
use \DateTime;

class Job
{
    ...

    public const OFFER_LIFETIME = 30; // durée de vie d'une offre en jours

    ...

    public function setExpiresAtValue(LifecycleEventArgs $event): self
    {
        // nous remplissons automatiquement la date d'expiration si cette dernière n'a pas été saisie
        // manuellement
        if (!$this->expiresAt) {
            $this->expiresAt = new DateTime('+' . self::OFFER_LIFETIME . ' day');
        }

        return $this;
    }
}
```



Symfony

QueryBuilder

- ❖ Il faut ensuite mettre à jour la configuration liée à cette gestion d'évènement **Job.orm.yml**

lifecycleCallbacks:

prePersist: [setCreatedAtValue, setExpiresAtValue] *# appelé lors de la création de l'entité*

preUpdate: [setUpdatedAtValue] *# appelé lors de la modification de l'entité*



Symfony

QueryBuilder

✦ Nous pouvons enfin mettre à jour la requête **JobRepository.php**

```
<?php
namespace App\Repository;

use Doctrine\ORM\EntityRepository;
use DateTime;

class JobRepository extends EntityRepository
{
    public function findActive()
    {
        return $this->createQueryBuilder('j')
            ->andWhere('j.expiresAt > :date')
            ->setParameter('date', new DateTime())
            ->getQuery()
            ->getResult();
    }
}
```

Ne pas oublier d'enlever le paramètre dans l'appel de la méthode dans la classe JobController::index



- ❏ Notre code est maintenant plus simple et plus maintenable. Mais si nous revenons à nos scénarios utilisateurs, nous avons spécifié que les offres devaient être classées par catégories, ce qui n'est actuellement pas le cas.
- ❏ Pour cela, il va falloir créer une classe de type Repository pour l'entité Category. C'est cette classe qui nous permettra de lister les catégories existantes avec les offres d'emploi correspondantes.



Symfony

QueryBuilder

❏ On commence par modifié le mapping Doctrine

```
# config/doctrine/mapping/Category.orm.yml  
App\Entity\Category:  
  type: entity  
  repositoryClass: App\Repository\CategoryRepository
```



❖ On créer ensuite la classe CategoryRepository.php

La requête Doctrine a été construite via le QueryBuilder. Doctrine implémente également son propre langage de requête appelé DQL.

Le QueryBuilder tout comme le DQL se base sur les entités que l'on a créées pour construire les requêtes effectuées en base de données.

Cela permet ensuite à Doctrine de créer les objets correspondants

```
<?php

namespace App\Repository;

use Doctrine\ORM\EntityRepository;

class CategoryRepository extends EntityRepository
{
    public function findCategoriesWithJobs()
    {
        return $this->createQueryBuilder('c')
            ->join('c.jobs', 'j')
            ->where('j.expiresAt >= :date')
            ->setParameter('date', new DateTime())
            ->getQuery()
            ->getResult();
    }
}
```



Symfony

QueryBuilder

🔧 On modifie ensuite notre contrôleur **JobController.php**

```
class JobController extends AbstractController
{
    /**
     * @Route("/job", name="job")
     */
    public function index(EntityManagerInterface $em):Response
    {
        $jobs = $em->getRepository(Category::class)->findCategoriesWithJobs();
        $categories = [];
        foreach ($categories as $category){
            $jobsCategories[$category->getName()] = $em->getRepository(Job::class)->findActiveByCategory($category);
        }
        return $this->render('job/index.html.twig', [
            'categories' => $jobsCategories,
        ]);
    }
}
```



- ❖ Il faut donc ajouter la méthode `findActiveByCategory` dans la classe **JobRepository.php**

```
class JobRepository extends EntityRepository
{
    public function findActiveByCategory(Category $category)
    {
        return $this->createQueryBuilder('j')
            ->where('j.category = :category')
            ->andWhere('j.expiresAt > :date')
            ->setParameter('category', $category)
            ->setParameter('date', new DateTime())
            ->getQuery()
            ->getResult();
    }
}
```



Symfony

QueryBuilder

🔧 Ensuite, nous modifions la vue en conséquence

```
{% block title %}Liste des offres d'emploi{% endblock %}

{% block body %}
    <h1 class="my-4">Liste des offres</h1>

    {% for category, Listjobs in categories %}
    <div class="row">
        <h2>{{ category }}</h2>

        {% for job in Listjobs %}
        <div class="row">
            <div class="col-5">
                <a href="#">
                    
                </a>
            </div>
            <div class="col-7">
                <h3>{{ job.position }}</h3>
                <p>{{ job.description }}</p>
                <p>Posté à {{ job.createdAt|date("m/d/Y") }}</p>
                <a class="btn btn-primary" href="{{ path('job_show', { 'id': job.id, 'company': job.companySlug, 'position': job.positionSlug }) }}">En savoir plus</a>
            </div>
        </div>
        <hr>
    {% endfor %}
    </div>
{% endfor %}
{% endblock %}
```

Symfony

QueryBuilder

❏ Pour finir, nous allons empêcher l'accès à une offre via son url si la date d'expiration est dépassée. Pour cela on rajoute un contrôle dans l'action d'affichage du contrôleur

```
<?php
namespace App\Controller;
...
use DateTime;

class JobController extends AbstractController
{
    ...
    public function show(EntityManagerInterface $em, int $id, string $company, string $position):Response
    {
        $job = $em->getRepository(Job::class)->find($id);
        if(null === $job){
            throw new NotFoundHttpException();
        }

        $currentDate = new DateTime();
        if($job->getExpiresAt() < $currentDate){
            throw new NotFoundHttpException();
        }

        return $this->render('job/show.html.twig', [
            'job' => $job,
        ]);
    }
}
```

Symfony

QueryBuilder

Recherche Emploi

Liste des offres

Programming



Web Developer

Vous avez déjà développé des sites Web avec symfony et vous souhaitez utiliser les technologies Open-Source. Vous avez au minimum 3 ans d'expérience dans le développement Web avec PHP ou Java et vous souhaitez participer au développement de sites Web 2.0 en utilisant les meilleurs frameworks disponibles.

Posté à 07/22/2019

[En savoir plus](#)

Design



Web Designer

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in.

Posté à 07/22/2019

[En savoir plus](#)

