TEK-UP Ecole Supérieure Privée Technologie & Ingénierie

Ateliers Framework (Symfony 6)

Wissem ELJAOUED
wissem.eljaoued@ensi-uma.tn

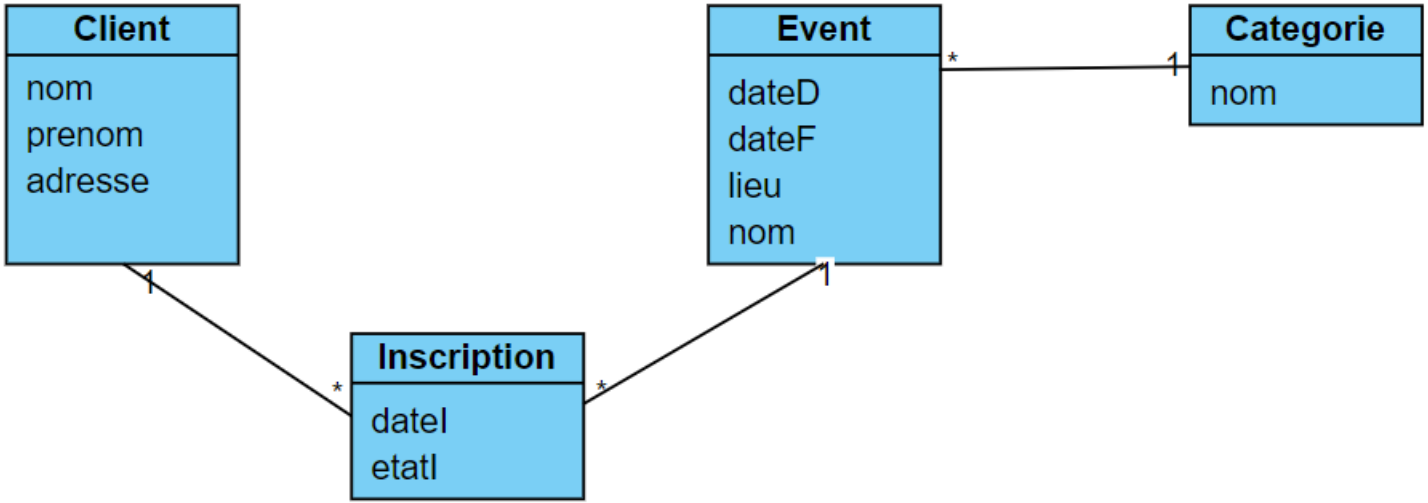
A.U. 2024-2025

Atelier 4

Doctrine

I.1. Etude de cas

- Nous allons travailler pendant ce TP avec l'étude de cas suivant:
  - « Gestion des évènements» représentée avec son diagramme de classes:



- Créer un nouveau projet intitulé « Events »

I.2. Création des entités

La commande suivante permet de créer une entité: symfony make:entity

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

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

 created: src/Entity/Event.php
 created: src/Repository/EventRepository.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):
 >dateD
 dateD

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

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

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

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

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

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

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

I.3. Configuration BD

- Les accès à la base de données ainsi qu'au serveur de messagerie sont centralisés dans le fichier App\.env

```
DATABASE_URL="mysql://root:@127.0.0.1:3306/events"
```

- Pour créer la base de données:

  symfony doctrine:database:create

I.4. Génération de la BD

Relations

Pour ajouter une relation entre l'entité Event et Categorie lancer la commande :

symfony make:entity Event

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

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

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

 What class should this entity be related to?:
 >Categorie
 Categorie
Categorie

 Is the Event.categorie property allowed to be null (nullable)? (yes/no) [yes]:
 >no
 no

 Do you want to add a new property to Categorie so that you can access/update Ev
ent objects from it - e.g. $categorie->getEvents()? (yes/no) [yes]:
 >yes
 yes
```

-1-

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

New field name inside Categorie [events]:
>events
events

Do you want to activate orphanRemoval on your relationship?
A Event is "orphaned" when it is removed from its related Categorie.
e.g. $categorie->removeEvent($event)

NOTE: If a Event may *change* from one Categorie to another, answer "no".

Do you want to automatically delete orphaned App\Entity\Event objects (orphanRe
moval)? (yes/no) [no]:
>no
no

updated: src/Entity/Event.php
updated: src/Entity/Categorie.php
```

-2-

Relations

- Ajouter les relations entre les entités:
    - Inscription ManyToOne Event
    - Inscription ManyToOne Client

I.4. Génération de la BD

- Pour faire le mapping entre les entités et la base de données, nous devons créer des migrations à l'aide de la commande suivante:

symfony make:migration

- Deuxièmement, exécuter ces migrations :

symfony doctrine:migrations:migrate

En cas d'erreur, changer l'URL de la base de données dans le fichier .env :

- en éliminant la version du serveur comme suit:

```
DATABASE_URL="mysql://root:@127.0.0.1:3306/workshop1
```

- Ou préciser la version exacte de votre serveur:

```
DATABASE_URL="mysql://root:@127.0.0.1:3306/workshop1?serverVersion=mariadb-10.4.14"
```

Contrôleur

- Créer un contrôleur EventController :
  Symfony make:controller EventController

- Ajouter la méthode listEvents

```php
#[Route('/event')]
class EventController extends AbstractController
{
    no usages
    #[Route('/', name: 'app_event')]
    public function listEvents(EventRepository $er): Response
    {
        $listEvents = $er->findAll();
        return $this->render( view: 'event/listEvents.html.twig',
            ['listeE'=>$listEvents]);
    }
}
```

- Créer le fichier listEvents.html.twig sous le répertoire event

```
<table border="2">
    <tr>
        <td>Date Debut</td>
        <td>Date Fin</td>
        <td>Lieu</td>
        <td>Nom</td>
        <td>Categorie</td>
    </tr>
    {% for e in listeE %}
        <tr>
            <td>{{ e.dateD | date }}</td>
            <td>{{ e.dateF | date }}</td>
            <td>{{ e.lieu }}</td>
            <td>{{ e.nom }}</td>
            <td>{{ e.categorie.nom }}</td>
        </tr>
    {% endfor %}
</table>
```

I.6. Ajout Form

Maintenant nous allons créer la classe formulaire, pour qu'il soit réutilisable là où on veut l'instancier . Pour cela il faut utiliser la commande suivante:

symfony make:form

```
 The name of the form class (e.g. OrangeChefType):
 >EventType
 EventType

 The name of Entity or fully qualified model class name that the new form will b
e bound to (empty for none):
 >Event
 Event
Event

 created: src/Form/EventType.php
```

I.6. Ajout Form

```
class EventType extends AbstractType
{
    no usages
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add( child: 'dateD',  type: null, [
                'widget' => 'single_text'
            ])
            ->add( child: 'dateF',  type: null, [
                'widget' => 'single_text'
            ])
            ->add( child: 'lieu')
            ->add( child: 'nom')
            ->add( child: 'categorie',  type: EntityType::class, [
                'class' => Categorie::class,
'choice_label' => 'id',
            ])
        ;
    }
```

- Créer l'action new dans le contrôleur EventController

```php
#[Route('/new', name: 'app_new')]
public function new(Request $request, EntityManagerInterface $em){
        $event = new Event();
        $form = $this->createForm( type: EventType::class, $event);
        $form->handleRequest($request);
        if($form->isSubmitted()){
            $em->persist($event);
            $em->flush();
            return $this->redirectToRoute( route: 'app_event');
        }
        return $this->render( view: 'event/new.html.twig',
        ['formE'=>$form->createView()]);
}
```

La variable $request contient les valeurs entrées dans le formulaire

Vérifier si la requête vient suite à un submit

Persister les données à travers l'ORM

Rediriger la page vers la liste des events suite à l'ajout

- Modifier la vue new.html.twig :

```
<form method="post">
    {{ form_widget(formE) }}
    <input type="submit" value="add Event">
</form>
```

Affichage du formulaire

Bouton de confirmation du formulaire

- Créer le lien vers l'action de suppression et ceci dans la vue de l'affichage

```
<table border="2">
    <tr>
        <td>Date Debut</td>
        <td>Date Fin</td>
        <td>Lieu</td>
        <td>Nom</td>
        <td>Categorie</td>
    </tr>
    {% for e in listeE %}
        <tr>
            <td>{{ e.dateD | date }}</td>
            <td>{{ e.dateF | date }}</td>
            <td>{{ e.lieu }}</td>
            <td>{{ e.nom }}</td>
            <td>{{ e.categorie.nom }}</td>
            <td><a href="{{ path('event_delete', {'id': e.id }) }}">Delete Event</a></td>
        </tr>
    {% endfor %}
</table>
```

Le paramètre à passer

La valeur du paramètre à passer

Lien vers la page de suppression (nom de la route)

- Maintenant il ne reste plus que l'action. Il faut récupérer l'ID passé en paramètre, récupérer l'entité ayant cet ID puis supprimer cette entité.

```php
#[Route('/{id}', name: 'event_delete')]
public function delete(EntityManagerInterface $em, EventRepository $er, $id){
        $event = $er->find($id);
        $em->remove($event);
        $em->flush();
 return $this->redirectToRoute( route: 'app_event');
}
```

Récupérer l'entité avec l'id spécifié

Supprimer l'entité récupérée et valider

Rediriger la page vers la liste des events

I.8. MAJ

- Pour la partie mise à jour elle est identique à celle de l'ajout, sauf qu'il faut seulement charger l'entité sélectionnée lors de l'appel de l'action update.

- Nous allons commencer par créer le lien vers le formulaire de mise à jour dans la view listEvents.html. twig

I.8. MAJ

```
<table border="2">
    <tr>
        <td>Date Debut</td>
        <td>Date Fin</td>
        <td>Lieu</td>
        <td>Nom</td>
        <td>Categorie</td>
    </tr>
    {% for e in listeE %}
        <tr>
            <td>{{ e.dateD | date }}</td>
            <td>{{ e.dateF | date }}</td>
            <td>{{ e.lieu }}</td>
            <td>{{ e.nom }}</td>
            <td>{{ e.categorie.nom }}</td>
            <td><a href="{{ path('event_delete', {'id': e.id }) }}">Delete Event</a></td>
            <td><a href="{{ path('event_update', {'id': e.id }) }}">Update Event</a></td>
        </tr>
    {% endfor %}
</table>
```

```php
#[Route('/{id}/edit', name: 'event_update')]
public function edit(Request $request, EntityManagerInterface $em, EventRepository $er, $id){
    $event = $er->find($id);

    $form = $this->createForm( type: EventType::class, $event);

    $form->handleRequest($request);

    if($form->isSubmitted()){
        $em->persist($event);
        $em->flush();
        return $this->redirectToRoute( route: 'app_event');
    }

    return $this->render( view: 'event/edit.html.twig',
        ['formE'=>$form->createView()]);
}
```

Charger l'instance qui correspond à l'ID en paramètre

- Créer la vue edit.html.twig :

```
<h1>Update Event</h1>

<form method="post">
    {{ form_widget(formE) }}
    <input type="submit" value="Update Event">
</form>
```