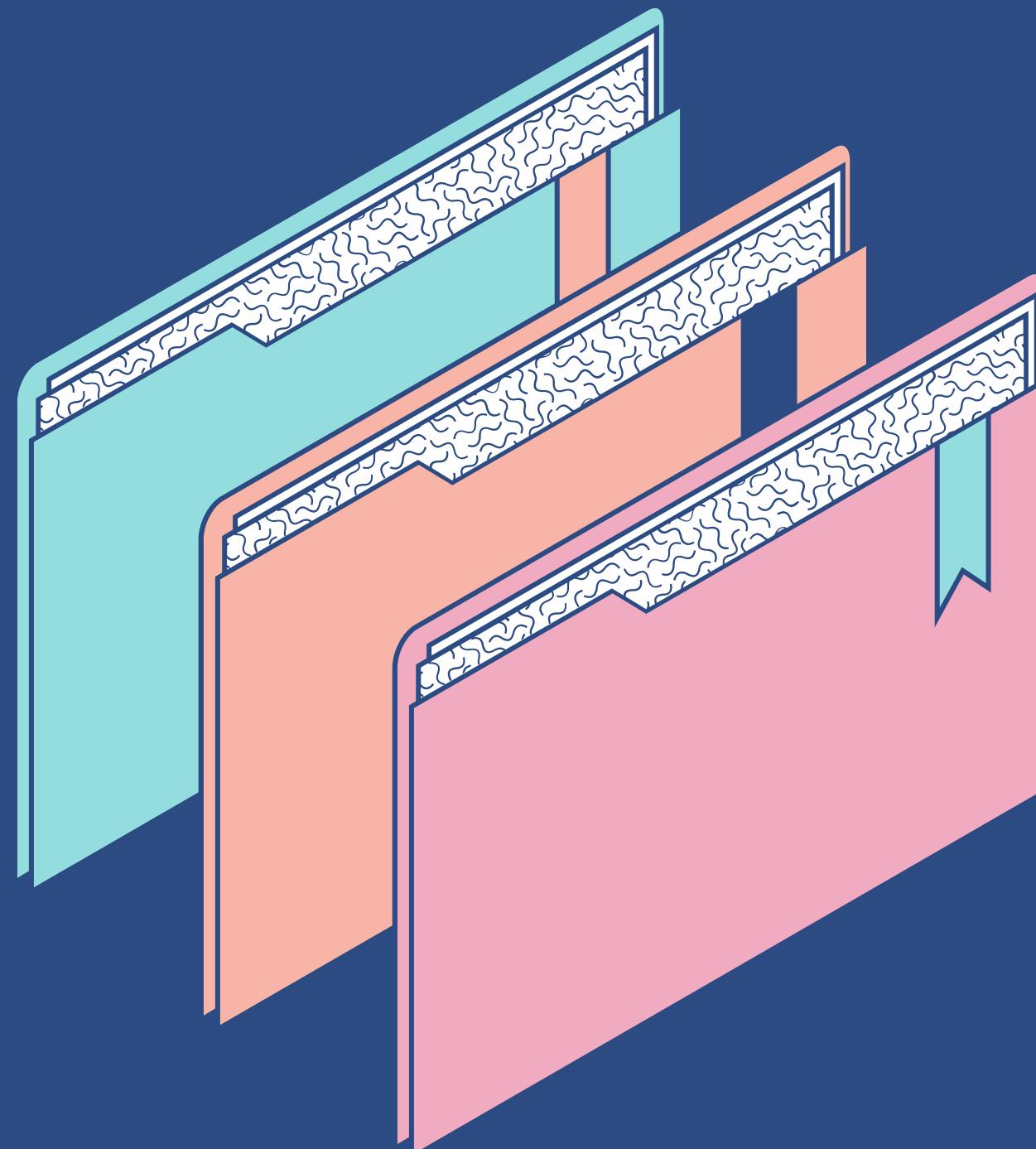




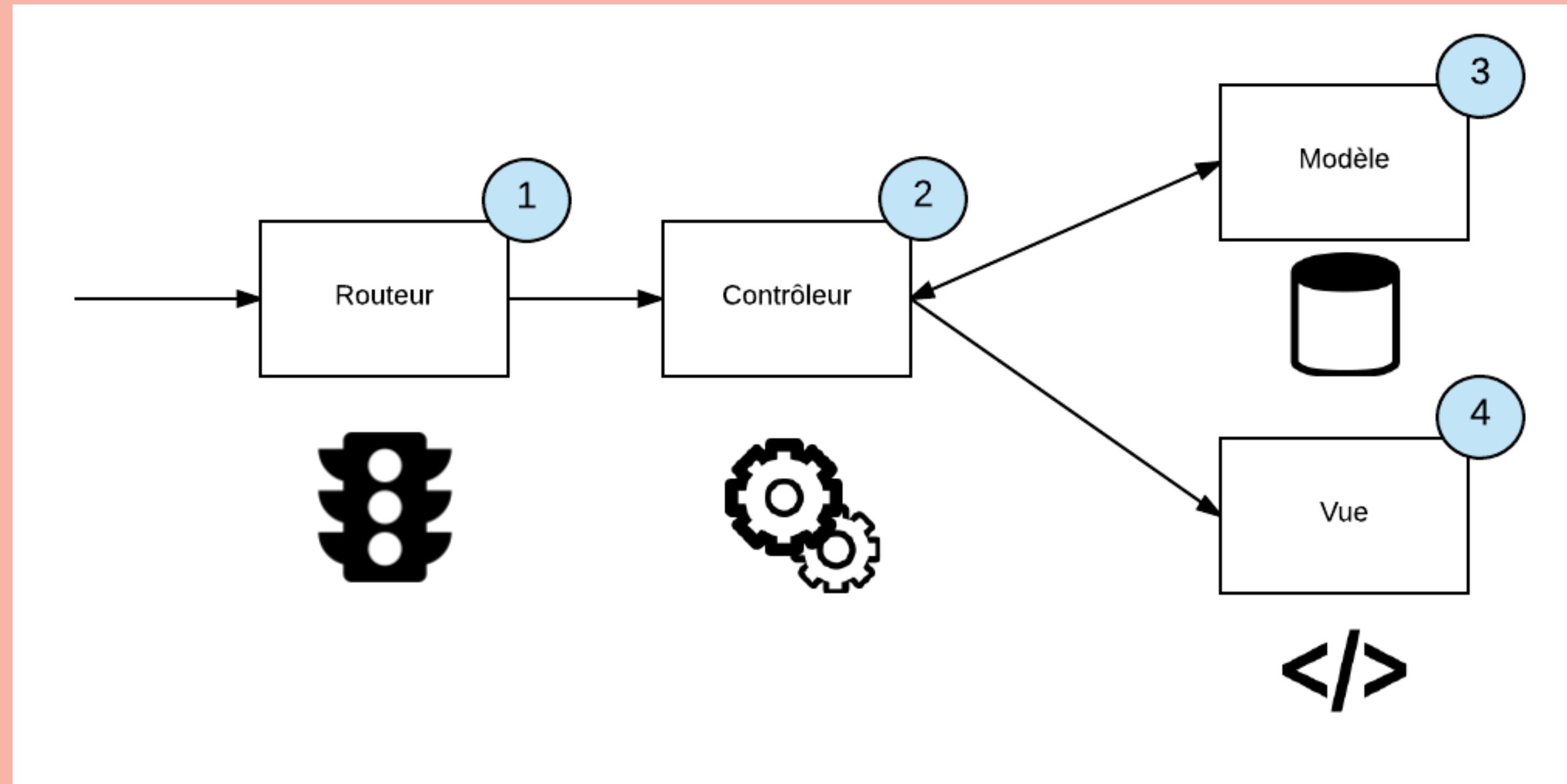
Symfony

SOMMAIRE



1. Modèles - Entités - ORM
2. Relation entre entités
3. Formulaires
4. Mails
5. Sécurité

Modèles - Entités - ORM



Introduction

Dans Symfony, la notion de modèle se retrouve sous la forme d'Entité.

Une entité est une classe PHP qui peut être connectée à une table de votre BDD via l'ORM (Object Relation Mapper).

Lorsqu'une entité est liée à une table, via l'ORM, il y aura un fichier "repository" associé. Il permet la génération de requêtes que le développeur peut modifier à volonté.

Introduction

Une ORM permet le CRUD des tables de données, remplaçant les requêtes SQL traditionnelles.

Symfony utilise l'ORM Doctrine par défaut, qui peut être géré en :
XML, JSON, YAML, PHP et Annotations.

La version la plus récente étant les annotations, c'est ce format que nous utiliserons par la suite.

Configuration

Pour fonctionner, il sera d'abord nécessaire d'installer les bundle ORM et maker:

```
composer require symfony/orm-pack
```

utilisation des orm

```
composer require symfony/maker-bundle --dev
```

Génération code console Symfony

Configuration

Une fois installé, il est nécessaire de configurer la connexion à la BDD.

Les informations user et pass doivent correspondre à vos logins habituels (PHPMyAdmin).

Le nom de la BDD peut être une BDD existante ou non

```
#DATABASE_URL="sqlite:///%kernel.project_dir%/var/data.db"  
DATABASE_URL="mysql://user:pass@127.0.0.1:3306/dbName"  
#DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
```



Création de la BDD

Une fois la configuration du .env effectué, nous pouvons créer la BDD :

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

Les modifications de structure de BDD devront être réalisées avec la console Symfony pour que les liens tables-ORM se réalisent



Création d'une entité liée à une table

On pourra créer ou modifier une entité à l'aide de la commande :

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

Vous devrez ensuite répondre à une liste de questions, avec le nom de l'entité (nom de la table), et les champs à créer.

En Symfony, une entité possède toujours un id PK AI, il ne faut pas le rajouter.



Création d'une entité liée à une table

Pour la création d'un champs, il vous faudra donner :

- son type
- sa taille (le cas échéant)
- si ce champs peut être null
- s'il doit être unique (index)

Le code de l'entité est ensuite généré dans "src/Entity/nomDeEntite.php"

Vous pouvez obtenir la liste des types disponibles en tapant "?" à la question du type

Mise à jour de la BDD

A ce stade, nous avons défini la configuration nécessaire pour notre BDD, nos tables et nos champs, mais nous n'avons pas encore appliqué cette configuration.

Pour exécuter ces instructions, nous allons créer une migration :

```
php bin/console make:migration
```

que nous exécuterons par la suite :

```
php bin/console doctrine:migrations:migrate
```

Modifications de champs et lien BDD

Pour modifier ou supprimer des champs, vous pouvez éditer directement le code de votre Entité générée dans : "src/Entity/nomDeEntite"

```
private ?int $id = null;  
  
#[ORM\Column(length: 255)]  
private ?string $zer = null;  
  
#[ORM\Column(type: Types::TIME_MUTABLE)]  
private ?\DateTimeInterface $size = null;
```



```
#[ORM\Column]  
private ?int $id = null;  
  
#[ORM\Column]  
private ?int $zer = null;  
  
#[ORM\Column(type: Types::TIME_MUTABLE)]  
private ?\DateTimeInterface $size = null;
```

Modifications de champs et lien BDD



Pour ajouter un champ, il sera nécessaire d'effectuer de nouveau la commande `make:migration` avec le nom de votre entité. Symfony détectera automatiquement que l'entité existe, et vous proposera d'ajouter vos champs.

Une fois les modifications effectuées, il faudra recréer une nouvelle migration, et la lancer pour que les modifications soient prise en compte.



ORM

Le CRUD de nos données s'effectuera maintenant de la même manière qu'en PHP POO natif.

Nous allons créer un nouveau Controller, qui permettra de définir la route selon laquelle nous effectuerons le traitement sur nos données.

Au sein de ce Controller, nous allons créer un nouvel objet correspondant à l'entité avec laquelle nous travaillons, et nous utiliserons le manager de classe pour effectuer le lien avec la BDD

ORM

```
use Doctrine\ORM\EntityManagerInterface;  
  
...  
  
#[Route("/test", name:"test")]  
public function test(EntityManagerInterface $entityManager)  
{  
    $post = new Post(); // initialise l'entité  
    $post->setTitle('Mon titre'); // on set les différents champs  
    $post->setEnable(true);  
    $post->setDateCreated(new \Datetime);  
  
    $entityManager->persist( $post ); // on déclare une modification de type persist et la générati  
    $entityManager->flush(); // on effectue les différentes modifications sur la base de données  
    // réelle  
  
    return new Response('Sauvegarde OK sur : ' . $post->getId() );  
}
```

On utilisera `$em->remove($post)` à la place de `persist` pour supprimer une donnée



Exercice 1

- Configurer une nouvelle BDD : `exo1Symfony`
- Créer la BDD
- Créer une entité Post, et ajoutez y les champs suivants :
 - titre, string 150
 - message, text
 - datePublication, datetime
- Créer un nouveau Controller "Post"
- Ajouter une route pour générer un nouvel enregistrement
 - Crée un objet Post et compléter les informations (titre, message, datePublication)
 - Récupérer la connexion à la doctrine (`$em = $this->getDoctrine()->getManager()`)
 - Associer l'instance de Post avec l'ORM (`$em->persist($post)`)
 - Enregistrer dans la base de données
- Appeler la route et vérifier le bon enregistrement de vos données

Recherche d'entité

Symfony et Doctrine propose des requêtes prédéfinies qui répondent aux usages les plus courant.

Si \$em est le manager associé à une entité :

- `$em->find($id);` on récupère qu'un seul élément de l'entité avec l'id \$id;
- `$em->findAll();` on récupère toutes les entrées de l'entité concernée
- `$em->findBy($where, $order, $limit, $offset);` on recherche avec le tableau \$where on tri avec le tableau \$order on récupère \$limit éléments à partir de l'élément \$offset.
- `$em->findOneBy($where, $order);` on récupère le premier élément respectant le tableau \$where et trié avec le tableau \$order;
- `$em->findByX($search);` requêtes magiques où X correspond à n'importe quel champs défini dans votre entité
- `$em->findOneByX($search) ;` requêtes magiques où X correspond à n'importe quel champs défini dans votre entité

Recherche d'entité

```
// Modifications multiples :  
#[Route("/est", name="test")]  
public function test(  
    EntityManagerInterface $entityManager,  
    PostRepository $postRepository)  
{  
    // récupération de tous les posts  
    $posts = $postRepository->findAll();  
    //équivalent à SELECT * FROM post  
  
    foreach($posts as $post)  
    {  
        $post->setTitle('Mon titre ' . $post->getId()); // on set les différents champs  
    }  
  
    $entityManager->flush(); // on effectue les différentes modifications sur la base de données  
    // réelle  
  
    return new Response('Sauvegarde OK');  
}
```



Recherche d'entité

Si aucune requête prédéfinie ne correspond à vos besoins, vous pouvez en écrire de nouvelles au sein du repository dédié à votre entité

```
// src/AppBundle/Repository/Post.php  
  
public function maRequeteSQL( $where )  
{  
    // avec requête SQL  
    $em = $this->getEntityManager();  
    $query = $em->createQuery('SELECT p from AppBundle:Post p  
WHERE p.title like :w');  
  
    $query->setParameter(':w', '%'.$where.'%');  
  
    return $query->getResult(); // on renvoie le résultat  
}
```

```
// src/AppBundle/Controller/DefaultController  
$postRepository->maRequete('test');
```

Modification

```

use App\Repository\PostRepository;
use Doctrine\ORM\EntityManagerInterface;

...

#[Route("/test/modification", name:"test")]
public function testModification(
    EntityManagerInterface $entityManager,
    PostRepository $postRepository)
{
    // récupération du post avec id 1
    $post = $postRepository->find(1);
    //équivalent à SELECT * FROM post WHERE id=1

    $post->setTitle('Mon titre'); // on set les différents champs
    $post->setEnable(true);
    $post->setDateCreated(new \Datetime);

    $entityManager->flush(); // on effectue les différentes modifications sur la base de données
    // réelle

    return new Response('Sauvegarde OK sur : ' . $post->getId());
}

```

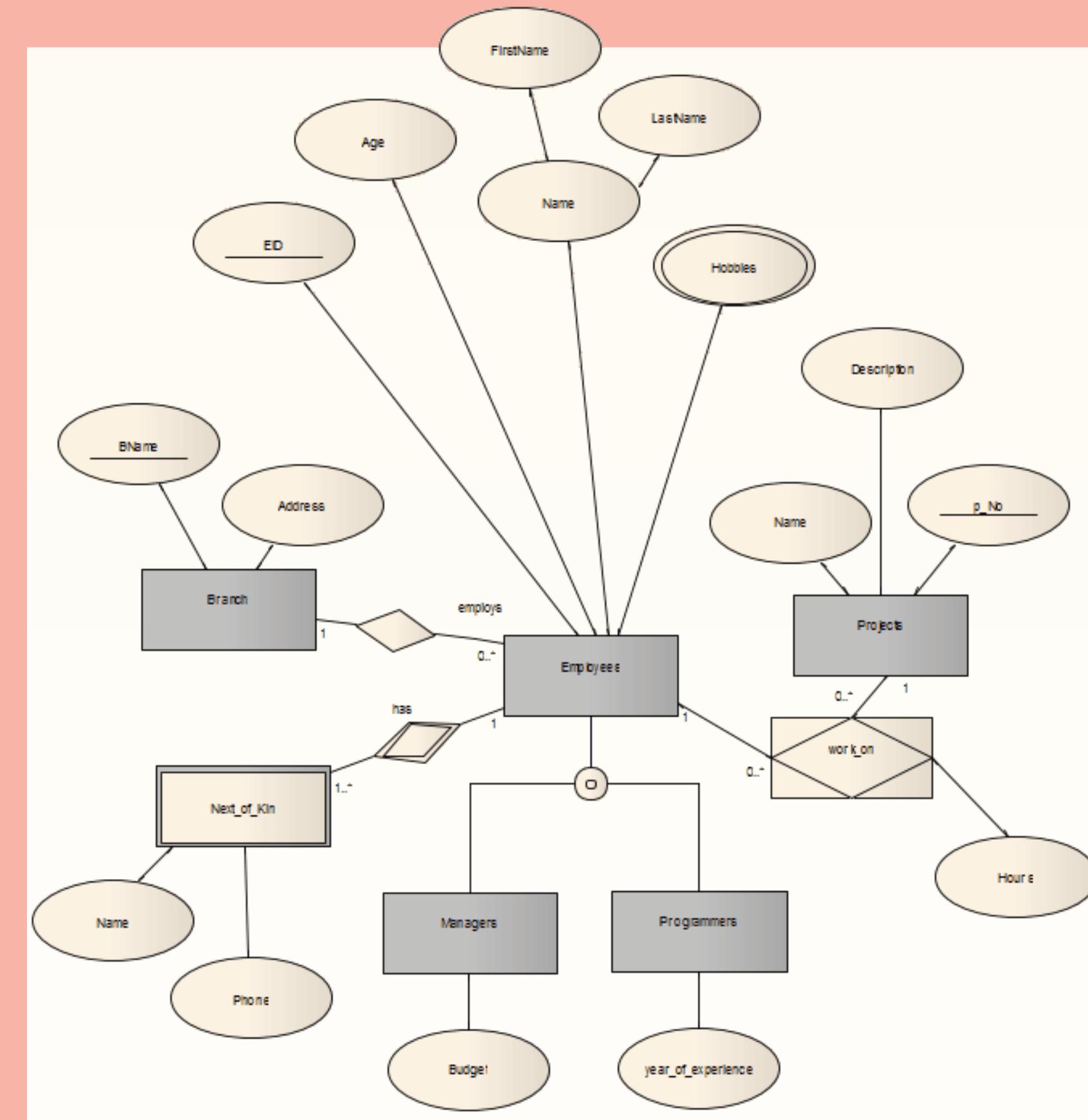
Pour modifier une entité, il suffit de l'instancier via une requête.

Le reste de la méthode est identique à la création.

Exercice 2

- Crée une nouvelle entité "PostCategory" avec :
 - title String 255
- Crée une page qui va sauvegarder une catégorie avec le nom Catégorie 1
- Crée une page qui va sauvegarder un post avec le nom Post 1, à la date courante, et avec pour message Lorem Ipsum
- Crée une page qui affiche le titre de la catégorie en id 1, et le post en id 1
- Crée un nouveau post identique au premier, en changeant le titre
- Crée une page qui affiche la totalité des entités Post

Relations entre entités



Introduction

Nous avons vu jusqu'à présent comment créer une BDD, des tables, et gérer le CRUD des données.

Cependant, nous n'avons pas encore abordé les relations entre les différentes entités (tables) via les clés étrangères

Dans une relation 1/n ou n/n, il y a toujours une entité propriétaire, et une entité inverse.
L'entité inverse fait référence à l'entité propriétaire.



Manipulation de la console

La console `make:entity` facilite grandement la création des relations.

En créant ou modifiant une entité, il est possible d'ajouter le champs contenant la relation en lui ajoutant un type "relation".

La console vous demandera de préciser l'entité liée, et le type de relation
(`ManyToOne`, `OneToMany`, `ManyToMany`, `OneToOne`)

Comme chaque modification, il faudra générer le fichier de migration et les appliquer

Manipulation de la console

```
php bin/console make:entity

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

to stop adding fields):
> category

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

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

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

Exercice 3

- Créer la liaison entre Post et PostCatégorie
- Modifier la page de génération de Post pour créer un nouveau post avec la relation vers "Catégorie 1"
 - La catégorie de ce post sera Catégorie 1
- Modifier la page avec tous les posts pour afficher la catégories liée à l'article
- Afficher la catégorie 1 avec tous les posts qui lui sont associés
- Créer une route qui supprime "Post 1"
- Créer une route qui supprime "Catégorie 1"

Formulaires



Introduction

La gestion des formulaire se fait via plusieurs classes PHP qui permettent entre autre :

- La structure et les propriétés du formulaire se gèrent via FormBuilder, et peuvent être réutilisées;
- On peut créer des classes spécifiques pour chacun de nos formulaires
- Permet une gestion des validations simplifiée et une sécurité renforcée
- Permet d'hydrater une entité ou un objet rapidement
- Gestion de template simple

Introduction

```
// src/Entity/Task.php
namespace App\Entity;

class Task
{
    protected $task;
    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }

    public function setTask($task)
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }

    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

Pour les exemples suivants, nous considérons l'entité ci-jointe

Introduction

Il existe 2 façons différentes de créer des formulaires :

- Directement au sein du Controller

Méthode la plus rapide à mettre en place (aucune dépendance ni fichier supplémentaire à mettre en place). Cependant, cela ne permet pas la réutilisation dans d'autres méthodes

- Via des classes dédiées FormType

Méthode nécessitant la mise en place de fichiers supplémentaires, mais permet une plus grande souplesse et une réutilisation dans d'autres contextes

Formulaire en Controller

```
public function new(Request $request)
{
    // Création d'un objet TASK
    $task = new Task();
    $task->setTask('Write a blog post');
    $task->setDate(new \DateTime('tomorrow'));

    // Création d'un formulaire avec données prédéfinies
    $form = $this->createFormBuilder($task)
        ->add('task', TextType::class)
        ->add('dueDate', DateType::class)
        ->getForm();

    // Envoi du formulaire à la vue
    return $this->render('default/new.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

Formulaire en FormType

Dans un premier temps, il sera nécessaire de créer notre nouvelle classe de formulaire.
Par convention, les modèles de formulaires sont créés dans : src/Form/

```
class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('task')
            ->add('dueDate', null, ['widget' => 'single_text'])
    }
}
```

Formulaire en FormType

Au sein de notre Controller, on pourra ensuite utiliser directement notre modèle de formulaire

```
class DefaultController extends AbstractController
{
    #[Route('/form', name: 'app_set_form')]
    public function new(): Response
    {
        // Création d'un objet TASK
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDate(new \DateTime('tomorrow'));

        // Création d'un formulaire avec données prédéfinies
        $form = $this->createForm(TaskType::class, $task);

        // Envoi du formulaire à la vue
        return $this->render('default/index.html.twig', [
            'form' => $form->createView(),
        ]);
    }
}
```

Formulaire TWIG

Une fois le formulaire créé et initié, il faut renvoyer le tout à TWIG via la méthode `createView`

```
return $this->render('template.html.twig', ['variable_form' => $form->createView()]);
```

Nous aurons ensuite de nombreuses fonctions TWIG utiles que nous pourrons utiliser :

- `{{ form_start(varForm) }}` permet de générer la balise `<form>` avec les différents attributs
- `{{ form_end(varForm) }}` permet de générer la fermeture de `<form>` avec les différents champs restants non affichés
- `{{ form_errors(varForm) }}` affiche les erreurs éventuelles du formulaire
- `{{ form_widget(mon formulaire.nomduchamps(varForm)) }}` affiche le type de champs
- `{{ form_label(mon formulaire.nomduchamps(varForm)) }}` affiche le label du champs

Formulaire TWIG

Ces fonctions permettent une grande maîtrise de la mise en forme de notre formulaire, mais demandent de détailler les éléments.

Il est également possible de d'afficher directement l'ensemble de notre formulaire à l'aide de la fonction `{{ form(variable_form) }}`

où `variable_form` correspond à la variable contenant le formulaire envoyée par le Controller

Formulaire TWIG

Il est enfin possible de préciser un thème pour notre formulaire avec la syntaxe :

```
{% form_theme form 'nom_du_template.html.twig' %}  
{{ form(variable_form) }}
```

https://symfony.com/doc/current/form/form_themes.html

Action / Request

Une fois le formulaire créé et affiché, nous devons ajouter le comportement à gérer à la soumission grâce aux méthodes :

- `handleRequest($request)` permet d'associer les valeurs input à la classe Form précédemment créé
- `isSubmitted()` permet de savoir si le formulaire a été envoyé
- `isValid()` permet de savoir si les données saisies sont valides

Action / Request

```
##[Route('/new', name: 'app_task_new', methods: ['GET', 'POST'])]
public function new(Request $request, TaskRepository $taskRepository): Response
{
    // Création du formulaire
    $task = new Task();
    $form = $this->createForm(Task1Type::class, $task);

    // récupération de la requête
    $form->handleRequest($request);
    // Test soumission et validité
    if ($form->isSubmitted() && $form->isValid()) {
        // Enregistrement de l'entité
        $taskRepository->save($task, true);
        // Redirection vers l'accueil
        return $this->redirectToRoute('app_task_index', [], Response::HTTP_SEE_OTHER);
    }

    return $this->renderForm('task/new.html.twig', [
        'task' => $task,
        'form' => $form,
    ]);
}
```

Validation

Les validations permettent de gérer des contraintes au niveau de la classe.

Il existe un grand nombre de validation possible (valeur, taille, type ...) détaillés dans la documentation ci-jointe.

De nombreux formats existent pour utiliser les validations
(XML,JSON,YAML,PHP,Annotations).

Nous privilégierons une fois de plus les annotations, étant la syntaxe la plus récente.

<https://symfony.com/doc/6.1/validation.html>

Validation

```
namespace App\Entity;  
// Ajout des validations  
use Symfony\Component\Validator\Constraints as Assert;  
use App\Repository\TaskRepository;  
use Doctrine\DBAL\Types\Types;  
use Doctrine\ORM\Mapping as ORM;  
  
#[ORM\Entity(repositoryClass: TaskRepository::class)]  
class Task  
{  
    #[ORM\Id]  
    #[ORM\GeneratedValue]  
    #[ORM\Column]  
    // Vérification non null sur le champ  
    #[Assert\NotBlank]  
    private ?int $id = null;
```



Génération du CRUD

Tout ce que nous venons d'effectuer manuellement peut être automatisé :

```
▶ symfony console make:crud
```

Vous devrez ensuite saisir le nom du bundle, le chemin du crud et l'ajout des fonctions d'ajout et de modification

**ATTENTION : La modification d'une entité ne modifiera pas les FormType.
Il faudra ajouter manuellement les champs créés !**

Exercice 4

- Générer le CRUD de Post avec l'url /admin/post
- Générer le CRUD de PostCategory avec l'url /admin/postcategory
- Tester le fonctionnement