Licence MIAGE **Université de Nantes**

Module X32I10P

Feuille de travaux pratiques

Série 4 : **Introduction à Symfony**

A) Premier pas :

Il s'agit d'utiliser **Symfony 4** pour développer une application web permettant de gérer les annonces de stages au sein d'une formation universitaire (exemple, L3 Miage).

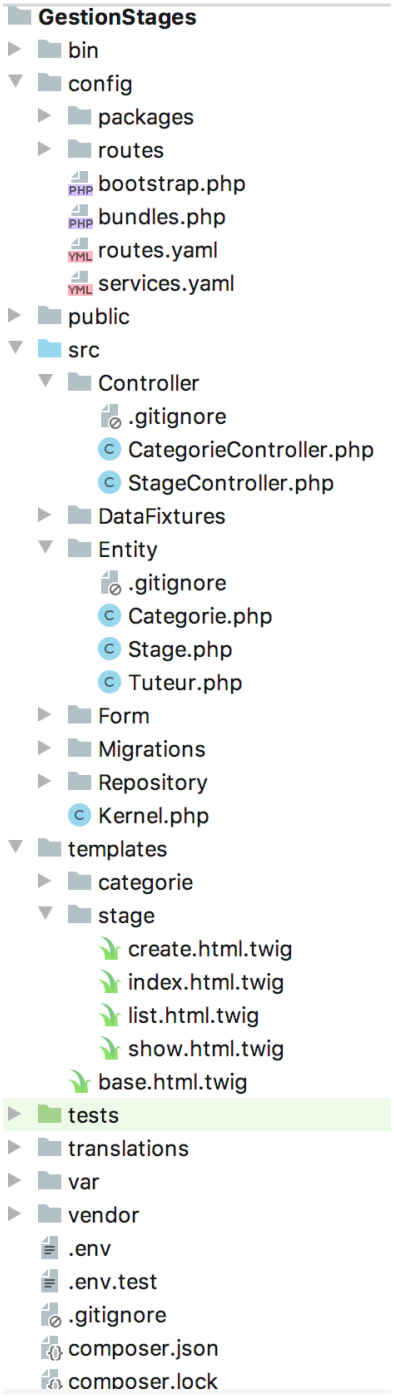
1) Utiliser composer pour créer un nouveau projet appelé *GestionStages* :

composer create-project symfony/website-skeleton:4.2.* GestionStages

Vérifier que la structure de votre projet correspond à l'organisation suivante :

Organisation générale d'une application Symfony 4.2

- **bin/** : contient les fichiers exécutables, en particulier la **console**.
- **config/** : contient les fichiers de configuration du projet.
- **public/** : contient le contrôleur frontal **index.php**, et tous les fichiers publiques comme le fichiers css, js, et les images.
- **src/** : contient tout le code PHP spécifique à l'utilisation de Symfony (contrôleurs et routes), les entités Doctrine et le code métier de l'application.
- **templates/**: contient les vues (twig, html, ...) de l'application.
- **tests/**: contient les tests de l'application.
- **translations/**: contient les fichiers de traduction.
- **var/cache/** : contient les fichiers de cache générés par l'application.
- **var/logs/** : contient les fichiers log générés par l'application.
- **vendor/** : contient les dépendances de l'application installées par *composer*.
- **.env** : le fichier contenant la configuration de l'environnement d'exécution de l'application (en particulier les paramètres de connexion à la base de données.



2) Vérifier le bon fonctionnement de l'application :

   a) En exécutant dans un terminal la commande suivante de la console :

      php bin/console list

   b) En lançant le serveur interne de php avec la commande suivante de la console :

      php bin/console server:run

   et en essayant dans un navigateur le lien suivant : http://127.0.0.1:8000

B) Persistance des données :

Un stage est décrit par un ensemble d'informations (poste, description, société, dates, ...) et appartient à une seule catégorie (Programmation, Administration du parc informatique, TMA, ...). En plus, il doit être encadré par au moins un tuteur de stage. Le modèle de données de l'application devra donc permettre de gérer les données des stages, des tuteurs et des catégories de stages.

1) On commence par paramétrer l'application pour qu'elle puisse accéder à une base de données. Pour cela, renseigner dans le fichier **.env** les paramètres nécessaires à l'établissement de la connexion.
2) Créer ensuite la base de données (si elle n'est existe pas) en exécutant dans un terminal la commande de la console suivante :

   php bin/console doctrine:database:create --if-not-exists

3) On va créer maintenant les classes, appelées **entités (Entity)**, associées au modèle de données de l'application. Ce sont des classes permettant de représenter les informations de la base de données sous forme d'objets PHP. Ces classes particulières sont à placer dans le dossier **src/Entity.**

   En utilisant la commande suivante de la console :

   php bin/console make:entity

   créer les trois entités suivantes :

   a) L'entité *Stage* décrite par les attributs suivants :

   - **id** : un entier unique permettant d'identifier un stage spécifique. Cet attribut sera automatiquement créé par Symfony pour toute classe entité.
   - **poste** : une chaîne de caractères de taille 255. Le poste pourrait être *développeur, analyste programmeur*, etc...
   - **description** : un texte décrivant le poste.
   - **type** : une chaîne de caractères de taille 255. Il y'a deux types possibles pour un stage : rémunéré ou non rémunéré.
   - **actif** : c'est un booléen pour indiquer si le stage est encore vacant.
   - **date_creation, date_expiration, date_modification** : respectivement la date de la création de l'annonce, la date de la dernière modification, la date d'expiration de l'annonce.
   - **email** : une adresse email pour le contact du stage.
   - **societe** : une chaîne de caractères de taille 255 contenant le nom de la société.
   - **ville** : une chaîne de caractères de taille 255.

b) L'entité *Categorie* décrite par les attributs suivants :

- **id** : un entier unique permettant d'identifier une catégorie spécifique.
- **nom** : une chaîne de caractères de taille 255 contenant le nom de la catégorie du stage. La catégorie pourrait être *Programmation, Administration du parc informatique, TMA, Design*.

c) L'entité *Tuteur* décrite par les attributs suivants :

- **id** : un entier unique permettant d'identifier un tuteur spécifique.
- **nom, prenom, email, telephone** : des chaînes de caractères décrivant un tuteur.

4) On va maintenant définir les relations entre ces 3 entités. On distingue en général quatre types de relation : **One-To-Many** (relation de type un vers plusieurs), **Many-To-One** (plusieurs vers un), **Many-To-Many** (relation de plusieurs à plusieurs) et **One-To-One**.

   En utilisant toujours la commande suivante de la console :

   php bin/console make:entity

   modifier l'entité Stage pour y ajouter :

   a) L'attribut ***categorie*** permettant d'établir une relation **ManyToOne** avec l'entité **Categorie.** Cette relation est à préciser quand la console demande de fournir le type du nouveau attribut.

   b) L'attribut ***tuteurs*** permettant d'établir une relation **ManyToMany** avec l'entité **Tuteur.**

5) Vérifier que la création des entités s'est bien passée en exécutant dans un terminal la commande suivante de la console :

   php bin/console doctrine:schema:validate

   Vérifier que les trois entités situées dans le dossier ***src/Entity/*** contiennent bien les attributs souhaités ainsi que leurs *annotations*, leurs *getters* et leurs *setters*.

6) On souhaite que, juste avant l'exécution de l'instruction de la persistance d'un objet de type *Stage*, les deux attributs **date_creation** et **date_modification** soient initialisés à la date courante. Pour ce faire, rajouter dans l'entité *Stage* les instructions suivantes écrites en gas :

```
/**
 * @ORM\Entity()
 * @ORM\Table(name="Stage")
 * @ORM\HasLifecycleCallbacks()
 * @ORM\Entity(repositoryClass="App\Repository\StageRepository")
 */
class Stage
{
...
/**
 * @ORM\PrePersist()
 */
public function prePersist()
{
    $this->date_creation = new \DateTime();
    $this->date_modification = new \DateTime();
}
/**
 * @ORM\PreUpdate()
 */
public function preUpdate()
{
    $this->date_modification = new \DateTime();
}
```

7) Créer le fichier de migration contenant les requêtes SQL de création des tables correspondantes aux entités. Pour ce faire, exécuter la commande de la console :

php bin/console doctrine:migration:diff

Qu'est ce que le fichier de migration dans le dossier **src/Migrations** contient ?

8) Exécuter le fichier de migration en utilisant la commande suivante de la console :

php bin/console doctrine:migration:migrate

9) Verifier (par exemple avec *phpMyAdmin*) que les tables ont bien été créées.

10) Avant de commencer l'implémentation des fonctionnalités de l'application, on va commencer par insérer un jeu de données initiales (également appelées **fixtures**). Pour ce faire, on va utiliser un bundle **DoctrineFixturesBundle**.

a) Télécharger avec composer le bundle **doctrine/doctrine-fixtures-bundle** :

composer require --dev doctrine/doctrine-fixtures-bundle:3.0.*

Quelle est la signification de l'option **--dev** de *Composer* ?

b) Placer dans le dossier **src/DataFixtures/** les fichiers **StageFixtures.php CategorieFixtures.php** et **TuteurFixtures.php** en ligne sur Madoc. Qu'est ce que ces fichiers contiennent ?

c) Exécuter la commande suivante de la console :

php bin/console doctrine:fixtures:load

d) Vérifier la présence des données dans la base de données avec :

php bin/console doctrine:query:sql "select * from Stage"

php bin/console doctrine:query:sql "select * from Categorie"

C) Les contrôleurs et les vues :

Symfony étant une implémentation du modèle de conception **MVC**, il répond à chaque requête du client en exécutant une méthode d'une classe appelée ***contrôleur***. Ce dernier récupère les données utilisateur pour y appliquer les traitements nécessaires et, si nécessaire, sollicite le **modèle** (***doctrine***) pour interroger une base de données. La réponse est envoyée par Symfony au client par le biais d'une **vue** (sous la forme d'une page Web, de données JSON/XML, etc...).

1) Créer un controlleur **StageController** en exécutant la commande de la console suivante :

php bin/console make:controller

Cette commande permet de créer le fichier **src/Controller/StageController.php**.

2) Le controlleur **StageController** contient les annotations suivantes :

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

Quelle est la signification de ces annotations ?

3) Lancer le serveur avec la commande de la console suivante :

php bin/console server:run

et essayer dans un navigateur le lien suivant : http://127.0.0.1:8000/stage. Quelle est la l'action exécutée par Symfony ? Quel est le fichier associé à la vue affichée ?

4) Pour afficher les détails de la route "stage", exécuter la commande de la console suivante :

php bin/console debug:router stage

5) Pour afficher la liste de toutes les routes définies dans l'application, exécuter la commande de la console suivante

php bin/console debug:router

6) Supprimer du controlleur **StageController** l'action **index (et ses annotations)** et y rajouter l'action **list** suivante :

```
/**
 * Lister tous les stages.
 * @Route("/stage/", name="stage.list")
 * @return Response
 */
public function list() : Response
{
    $stages = $this->getDoctrine()->getRepository(Stage::class)->findAll();

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

Qu'est ce qu'elle permet cette action ?

7) Télécharger sur Madoc :

   1) Le template **base.html.twig** pour remplacer celui dans le dossier **templates**.
   2) La vue **list.html.twig** à placer dans le dossier **templates/stage/.**

8) L'exécution dans un navigateur du lien : http://127.0.0.1:8000/stage génère quelques exceptions. Quelles sont les raisons ? Corriger ces erreurs.

9) Rajouter dans **StageController** l'action **show** suivante :

```
/**
 * Chercher et afficher un stage.
 * @Route("/stage/{id}", name="stage.show", requirements={"id" = "\d+"})
 * @param Stage $stage
 * @return Response
 */
public function show(Stage $stage) : Response
{
    return $this->render('stage/show.html.twig', [
        'stage' => $stage,
    ]);
}
```

10) Placer dans le dossier **templates/stage/** la vue **show.html.twig** (en ligne sur Madoc), et modifier les liens dans la vue **list.html.twig** :

```
<a href="{{ path('stage.show', {id: stage.id}) }}">{{ stage.poste }}</a>
```

et la vue **base.html.twig** :

```
<a class="navbar-brand" href="{{ path('stage.list') }}">Offres des stages</a>
```

11) On souhaite insérer dans la base de données un jeu de données contenant des stages expirés. Pour ce faire, modifier le fichier ***StageFixtures*** pour pouvoir ajouter deux stages dont la date d'expiration est passé il y'a un mois. Exécuter la commande suivante de la console :

php bin/console doctrine:fixtures:load

NB : Noter que la commande au-dessus écrase les données existantes dans les tables associées aux entités concernées.

12) Changer dans **StageController** l'action **show** pour ne récupérer que les stages qui n'ont pas encore expiré :

```
/**
 * Lister uniquement les stages qui n'on pas encore expiré !
 * @Route("/stage", name="stage.list")
 * @return Response
 */
public function list(EntityManagerInterface $em) : Response
{
    $query = $em->createQuery(
        'SELECT s FROM App:Stage s WHERE s.date_expiration > :date'
    )->setParameter('date', new \DateTime());
    $stages = $query->getResult();

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

Vérifier que la vue n'affiche que des stages encore valides.

13) On souhaite que, juste avant l'exécution de l'instruction de la persistance d'un objet *Stage*, si la date d'expiration du stage n'a pas été fourni par l'utilisateur, cette date doit être automatiquement initialisée à 20 jours. Pour ce faire, rajouter dans la méthode **prePersist** de l'entité **Stage** les instructions suivantes écrites en gas :

```
/**
 * @ORM\PrePersist()
 */
public function prePersist()
{
    ...
    if (!$this->date_expiration) {
        $this->date_expiration = (clone $this->date_creation)->modify('+20 days');
    }
}
```

14) L'action **show** du controlleur **StageController** utilise des requêtes SQL pour chercher les stages qui n'ont pas encore expiré. Une bonne pratique dans l'utilisation de Symfony consiste à déplacer ces requêtes dans la classe **StageRepository**. Rajouter dans cette classe la méthode suivante :

```
/**
 * @return Stage[]
 */
public function getStagesNonExpires()
{
    $qb = $this->createQueryBuilder('s')
        ->where('s.date_expiration > :date')
        ->setParameter('date', new \DateTime())
        ->orderBy('s.date_creation', 'DESC');

    return $qb->getQuery()->getResult();
}
```

Modifier l'action **list** en faisant appel à la méthode au-dessus.

15) On souhaite maintenant afficher les stages selon leur catégorie (Programmation, TMA, ...). Pour ce faire :

a) En utilisant la console de Symfony, créer un nouveau controlleur **CategorieController.**

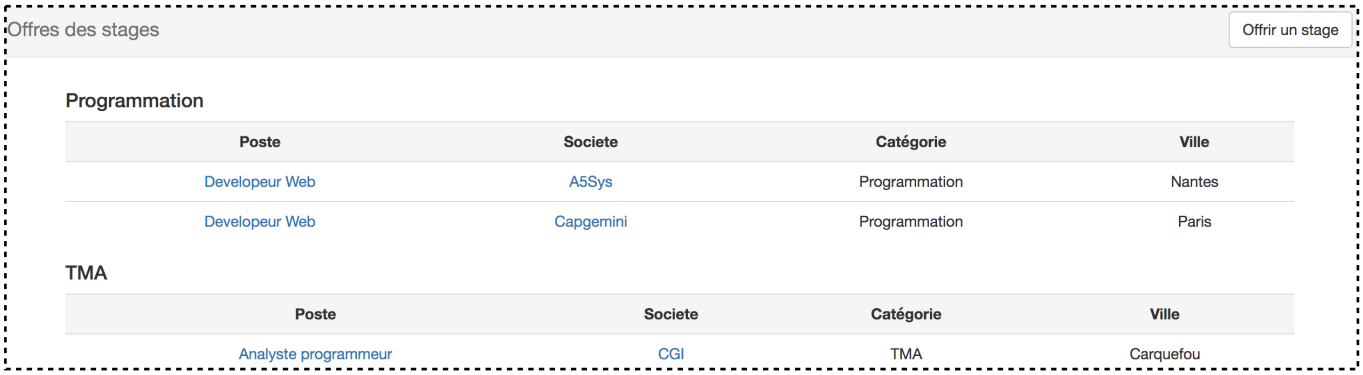
b) Rajouter dans l'entité **Categorie** la méthode **getStagesNonExpires** permettant de ne retourner que les stages non expirés d'une catégorie donnée. Cette nouvelle méthode est une alternative au ***getter*** getStages.

c) Rajouter dans **CategorieRepository** la méthode suivante :

```
/**
 * @return Categorie[]
 */
public function getCategoriesAvecStagesNonExpires()
{
    return $this->createQueryBuilder('c')
        ->select('c')
        ->innerJoin('c.stages', 's')
        ->where('s.date_expiration > :date')
        ->setParameter('date', new \DateTime())
        ->getQuery()
        ->getResult();
}
```

permettant de chercher dans la base de données les catégories de stages pour lesquels il y'a des annonces de stage non expirées.

d) Utiliser la méthode au-dessus dans l'action **list** du controlleur **CategorieController.** Cette action devra rendre une vue categorie/list.html.twig illustrée par :



16) Pour utiliser des URLs optimisés pour les moteurs de recherche, on va générer pour chaque stage un "***slug***" à partir de son attribut poste. Le *slug* va être utilisé pour identifier un stage au lieu d'utiliser son attribut *id*. Pour ce faire :

a) Télécharger avec composer le package **stof/doctrine-extensions-bundle**

b) Modifier **config/packages/stof_doctrine_extensions.yaml** comme suit :

```
stof_doctrine_extensions:
    default_locale: en_US
    orm:
        default:
            sluggable: true
```

c) Ajouter dans l'entité **Stage** un attribut **slug** et ses *getter/setter* :

```
....
use Gedmo\Mapping\Annotation as Gedmo;
....

class Stage
{
...

    /**
    * @var string
    * @Gedmo\Slug(fields={"poste"})
    * @ORM\Column(type="string", length=128, unique=true)
    */
    private $slug;

    /**
     * @return string|null
     */
    public function getSlug() : ?string
    {
        return $this->slug;
    }

    /**
     * @param string $slug
     */
    public function setSlug(string $slug): void
    {
        $this->slug = $slug;
    }
```

d) Changer les annotations de l'action **show** du controlleur **StageController** comme suit **:**

```
 * @Route("/stage/{slug}", name="stage.show")
 * @param Stage $stage
 * @return Response
 */
public function show(Stage $stage) : Response
{
...
```

- Mettre à jour le schéma de la base de données en exécutant :

php bin/console doctrine:migrations:diff

php bin/console doctrine:schema:drop --force --full-database

php bin/console doctrine:migration:migrate

php bin/console doctrine:fixtures:load

- Mettre à jour les liens des dans les vues.
- Vérifier le bon fonctionnement en allant sur http://127.0.0.1:8000/

D) Les formulaires :

Pour pouvoir créer ou éditer une annonce d'un stage, on a besoin d'utiliser des formulaires. Symfony permet de générer des formulaires de création/édition d'une entité à partir de sa définition. Dans ce qui suit, on s'intéresse aux formulaires de création/édition de l'entité **Stage**.

1) Créer la classe **StageType** en exécutant la commande de la console suivante :

php bin/console make:form

Cette commande permet de créer le fichier **src/Form/StageType.php**.

2) Modifier la méthode **buildForm** dans la classe **StageType** comme suit :

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('poste', TextType::class)
        ->add('type',ChoiceType::class, [
            'choices'  => [ 'Oui' => true,'Non' => false,],
            'label' => 'Plein temps ?'])
        ->add('description', TextareaType::class)
        ->add('actif',ChoiceType::class, [
            'choices'  => [
                'Oui' => true,'Non' => false, ],
                'label' => 'Actif ?'])
        ->add('email', EmailType::class)
        ->add('societe', TextType::class)
        ->add('ville', TextType::class)
        ->add('categorie', EntityType::class, [
            'class' => Categorie::class, 'choice_label' => 'nom',])
        ->add('tuteurs', EntityType::class, [
            'class' => Tuteur::class,
            'multiple' => true,
            'expanded' => true,
            'choice_label' => 'nom',
    ]);
}
```

3) Rajouter dans **StageController** l'action **create** :

```
/**
 * Créer un nouveau stage.
 * @Route("/nouveau-stage", name="stage.create")
 * @param Request $request
 * @param EntityManagerInterface $em
 * @return RedirectResponse|Response
 */
public function create(Request $request, EntityManagerInterface $em) : Response
{
    $stage = new Stage();
    $form = $this->createForm(StageType::class, $stage);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $em->persist($stage);
        $em->flush();
        return $this->redirectToRoute('stage.list');
    }
    return $this->render('stage/create.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

4) Télécharger sur Madoc la vue **templates/stage/create.html.twig.**

5) En utilisant la fonction ***path*** de *twig*, ajouter le lien de création de stage à votre application. Essayer de créer un nouveau stage.

6) Pour utiliser **bootstrap** dans le formulaire de création de stage, rajouter dans **config/packages/twig.yaml** les lignes suivantes :

```
twig:
...
    form_themes:
        - 'bootstrap_3_horizontal_layout.html.twig'
```

7) Modifier la méthode **buildForm** dans la classe **StageType** pour rajouter les contraintes suivantes :

```
....
->add('societe', TextType::class, [
    'constraints' => [new NotBlank(),
        new Length(['max' => 255]),
        ]
])
....
```

Quelle est la signification de ces contraintes ? Rajouter ces contraintes à d'autres champs du formulaire.

8) Pour pouvoir éditer une annonce d'un stage :

a) Rajouter dans **StageController** l'action **edit :**

```
/**
 * Éditer un  stage.
 * @Route("stage/{slug}/edit", name="stage.edit")
 * @param Request $request
 * @param EntityManagerInterface $em
 * @return RedirectResponse|Response
 */
public function edit(Request $request, Stage $stage, EntityManagerInterface $em) : Response
{
    $form = $this->createForm(StageType::class, $stage);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $em->flush();
        return $this->redirectToRoute('stage.list');
    }

    return $this->render('stage/create.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

NB : la vue du formulaire d'édition d'un stage est la même que celle de la création (c.à.d *stage/create.html.twig).* Si nécessaire, il faut prévoir une vue différente.

b) En utilisant la fonction ***path*** de *twig*, ajouter dans la vue **templates/stage/list.html.twig** un lien d'édition pour chaque stage.

c) Vérifier le bon fonctionnement de l'action d'édition d'un stage.

9) Pour pouvoir supprimer une annonce d'un stage :

a) Rajouter dans **StageController** l'action **delete :**

```
/**
 * Supprimer un stage.
 * @Route("stage/{slug}/delete", name="stage.delete")
 * @param Request $request
 * @param Stage $stage
 * @param EntityManagerInterface $em
 * @return Response
 */
public function delete(Request $request, Stage $stage, EntityManagerInterface $em) : Response
{
    $form = $this->createFormBuilder()
        ->setAction($this->generateUrl('stage.delete', ['slug' => $stage->getSlug()]))
        ->getForm();

    $form->handleRequest($request);
   if ( ! $form->isSubmitted() || ! $form->isValid()) {
        return $this->render('stage/delete.html.twig', [
        'stage' => $stage,
        'form' => $form->createView(),
    ]);
    }

    $em = $this->getDoctrine()->getManager();
    $em->remove($stage);
    $em->flush();
    return $this->redirectToRoute('stage.list');
}
```

b) Placer dans le dossier **templates/stage/** la vue **delete.html.twig** (en ligne sur sur Madoc).

c) En utilisant la fonction ***path*** de *twig*, ajouter dans la vue **templates/stage/list.html.twig** un lien de suppression pour chaque stage.

d) Vérifier le bon fonctionnement de l'action de suppression d'un stage.

J) Administration de l'application :

On souhaite rajouter dans l'application une couche permettant d'effectuer certaines opérations spécifiques à l'administration. Cette couche permettra par exemple de rajouter dans la base de données des nouvelles catégories de stages et des nouveaux tuteurs. Pour ce faire, deux choix sont possibles :

- Procéder de la même façon que ce qui était fait pour la gestion des stages : créer un nouveau contrôleur pour les opérations d'administration avec toutes les vues associées.
- Utiliser un bundle d'administration qu'on peut intégrer dans l'application. C'est cette option qui est recommandé et qu'on va utiliser dans la suite de cette série.

1) On va utiliser le bundle ***easycorp/easyadmin-bundle*** pour l'administration de l'application. On peut utiliser *composer* pour le télécharger, mais son intégration dans l'application nécessite aussi la création d'un fichier de configuration. Une alternative intéressante est d'utiliser ***Symfony flex*** pour une intégration facile du nouveau bundle. Pour ce faire, exécuter la commande suivante de la console :

composer req admin

2) Rajouter l'entité ***Categorie*** dans la liste des entités administrées par le bundle admin définie dans le fichier **config/packages/easy_admin.yaml**.
3) Créer une nouvelle catégorie de stage en utilisant la nouvelle route d'administration.
4) Créer un nouveau stage appartenant à la nouvelle catégorie.
5) Utiliser ce bundle pour l'administration des tuteurs aussi.

NB : Ce bundle est à utiliser exclusivement pour la configuration de l'application (ajout de nouvelles catégories de stage...). Il ne faut pas l'utiliser pour toutes les opérations CRUD de l'application.

F) Sécurité de l'application :

En utilisant Symfony, on souhaite limiter l'accès à certaines fonctionnalités de l'application à certains profiles, appelés ***rôles***, d'utilisateurs. Pour ce faire, Symfony se base sur un **système d'authentification** et un **système d'authorisation**.

1) Le *système d'authentification* est basé sur l'utilisation du bundle **symfony/security-bundle**. Pour le télécharger, exécuter la commande suivante de la console :

composer require symfony/security-bundle

2) Créer la classe ***User*** permettant l'authentication d'un utilisateur. Pour ce faire, exécuter la commande suivante de la console :

php bin/console make:user

On assume qu'un utilisateur sera identifié par son *email*.

3) La politique de sécurité de l'application est définie dans le fichier **config/packages/security.yaml**. Vérifier que Symfony a ajouté dans ce fichier les annotations suivantes permettant d'utiliser lors de l'authentification la classe ***User*** :

```
app_user_provider:
    entity:
        class: App\Entity\User
        property: email
```

Symfony a aussi ajouté les instructions suivantes pour préciser l'algorithme de ***cryptage*** (par défaut ***bcrypt***) pour les mots de passe :

```
encoders:
    App\Entity\User:
        algorithm: bcrypt
```

4) Télécharger le fichier **UserFixtures.php** en ligne sur Madoc et exécuter la commande de la console pour insérer dans la base de données les utilisateurs définis dans ce fichier.

5) Le *système d'authorisation* est basé sur la définition des rôles d'utilisateurs. Vérifier que Symfony a bien défini un attribut **roles** dans la classe *User* comme le montrent les instructions suivantes :

```
// src/Entity/User.php
// ...

/**
 * @ORM\Column(type="json")
 */
private $roles = [];

public function getRoles(): array
{
    $roles = $this->roles;
    // guarantee every user at least has ROLE_USER
    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}
```

6) Pour générer le formulaire d'authentification :

   a) Exécuter la commande suivante de la console :

   php bin/console make:auth

   Appeler la classe d'authentification avec le nom *LoginFormAuthenticator*. Ceci permet de créer le fichier ***app/Security/LoginFormAuthenticator.php***

   b) Dans la classe ***LoginFormAuthenticator***, modifier la méthode ***onAuthenticationSuccess*** pour définir une redirection vers la route ***stage.list*** quand l'utilisateur se déconnecte :

```
// src/Entity/User.php
// ...

public function onAuthenticationSuccess(Request $request, TokenInterface
$token, $providerKey)
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
        return new RedirectResponse($targetPath);
    }

    return new RedirectResponse($this->urlGenerator->generate('stage.list'));
}
```

7) On souhaite sécuriser l'accès aux fonctionnalités de l'application et exiger que les utilisateurs aient des rôles spécifiques. Pour ce faire, il y'a deux méthodes possibles :

   a) **Méthode 1** : par exemple, pour limiter l'accès aux fonctionnalités de l'administration aux utilisateurs ayant le rôle ***ROLE_ADMIN,*** rajouter dans le fichier **config/packages/security.yaml** les annotations suivantes :

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

    firewalls:
        # ...
        main:
            # ...

    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: ^/admin, roles: ROLE_ADMIN }
```

   Tester le bon fonctionnement du système d'authentification en asseyant une fonctionnalité de l'administration de l'application, par exemple la création d'une nouvelle catégorie de stage.

   b) **Méthode 2** : par exemple, pour exiger que l'utilisateur s'authentifie avant qu'il puisse exécuter les actions du contrôleur *StageController*, rajouter dans le fichier **StageController.php** les annotations suivantes :

```
/**
 * Require ROLE_USER for *every* controller method in this class.
 *
 * @IsGranted("ROLE_USER")
 */

class StageController extends AbstractController
{
....
```

8) Quand l'utilisateur essaye d'exécuter une action sans qu'il possède le rôle nécessaire, Symfony déclenche l'exception ***AccessDeniedException***. Pour personnaliser la gestion de cette exception :

   a) Télécharger sur Madoc le contrôleur **AccessDeniedController**.

   b) Configurer le système de sécurité de l'application pour déléguer la gestion de l'exception *AccessDeniedException* à ce contrôleur. Pour ce faire, rajouter les annotations suivantes :

```
main:
    ....
    access_denied_handler: App\Controller\AccessDeniedController
```

I) Les services Symfony :

Pour faciliter la navigation dans l'application, les trois dernières annonces consultées par l'utilisateur doivent être affichées dans le menu. Pour ce faire, lorsqu'un utilisateur consulte les détails d'un stage, l'objet de stage affiché doit être ajouté dans l'historique utilisateur et stocké dans la session.

1) Créez un nouveau **service *StageHistoryService*** en plaçant dans le dossier src/Service la classe **StageHistoryService** en ligne sur Madoc.

2) Pour garder une trace des consultations de l'utilisateur, modifier la méthode **show** dans le contrôleur **StageController** comme suit :

```
....
/**
 * chercher et affiche un stage.
 * @Route("/stage/{slug}", name="stage.show", methods="GET")
 * @param Stage $stage
* @return Response
 */
public function show(Stage $stage, StageHistoryService $stageHistoryService) : Response
{
    $stageHistoryService->addStage($stage);
    return $this->render('stage/show.html.twig', [
        'stage' => $stage,
    ]);
}
....
```

3) Pour afficher les 3 derniers stages consultés par l'utilisateur, modifier la méthode **list** dans le contrôleur **StageController** comme suit :

```
...
/**
  * Lister uniquement les stages qui n'on pas encore expiré !
  * @Route("/stage", name="stage.list")
  * @param StageHistoryService $stageHistoryService
  * @return Response
*/
    public function list(EntityManagerInterface $em, StageHistoryService
$stageHistoryService) : Response
    {
        $stages = $em->getRepository(Stage::class)->getStagesNonExpires();

        return $this->render('stage/list.html.twig', [
            'stages' => $stages,
            'historyStages' => $stageHistoryService->getStages(),
        ]);
    }
...
```

4) Modifier la vue **list.html.twig** pour afficher en haut de la page les 3 derniers stages consultés par l'utilisateur come suit :

J) La traduction dans Symfony :

1) Dans la vue base.html.twig, modifier le label du bouton de création de stage comme suit :

```
<a href="{{ path('stage.create') }}" class="btn btn-success navbar-btn">{{'stage.offrir'|trans}}</a>
```

2) On va générer les fichiers en format *yaml* que Symfony utilise dans la traduction de l'application. Cette dernière sera traduite en français et en anglais. La génération de ces fichiers peut se faire avec la commande suivante de la console :

```
php bin/console translation:update --dump-messages --force --output-format yaml fr
php bin/console translation:update --dump-messages --force --output-format yaml en
```

3) Modifier les fichiers générés dans le dossier ***translations*** pour traduire le mot clé stage.offrir.

4) La langue par défaut utilisée par symfony est définie dans le fichier **config/packages/translation.yaml**.

```
# config/packages/translation.yaml
framework:
    default_locale: en
    translator:
        # ...
```

En changeant la langue par défaut, vérifier le bon fonctionnement de la traduction.

5) On souhaite que les URLs de toutes les routes définies dans la contrôleur ***StageController*** aient un préfix **fr** (respectivement **en**) quand l'application est traduite en français (respectivement anglais). Pour ce faire, rajouter au début de cette classe les instructions suivantes en gras :

```
  * ....
  * @Route("/{_locale}/")
 */

class StageController extends AbstractController
{
...
```

En utilisant la console, vérifier que toutes les routes ont le préfixe ***/{_locale}/***

6) On souhaite que l'application puisse être traduite dans différentes langues qu'on pourrait paramétrer. Ces langues utilisées doivent faire partie des paramètres de l'application. Pour ce faire :

   a) Définir la **variable globale languages** dans le fichier **config/services.yaml** :

```
....
parameters:
   languages: {'fr':'Français','en':'English','de':'Deutsch'}

....
```

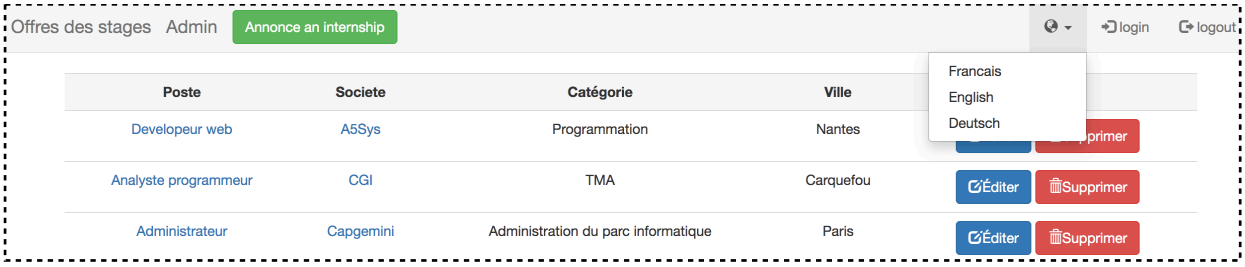
   b) Rendre cette variables accessible dans les templates Twig. Pour ce faire, définir une variable **locales** dans le fichier **config/packages/twig.yaml** :

```
twig:
  ...
   globals:
      locales: '%languages%'
```

   c) Utiliser la variable **locales** dans le template **base.html.twig** :pour afficher des liens permettant de choisir la langue de traduction. Par exemple, rajouter la boucle suivante :

```
....
<li class="dropdown">
   <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-
haspopup="true" aria-expanded="false">
      <span class="glyphicon glyphicon-globe"></span> <span class="caret"></span></a>
   <ul class="dropdown-menu">
      {% for key,locale in locales %}
         <li><a href="{{ path(app.request.get('_route'),
app.request.attributes.get('_route_params')|merge({'_locale': key})) }}">{{locale}}</a></li>
      {% endfor %}
   </ul>
</li>
....
```

La barre de navigation pourrait ressembler à la capture d'écran suivante :



   d) Modifier le fichier translations.yaml pour que la langue par défaut et celle de replis (fallbacks) soit une variable globale.

J) Une nouvelle commande de la console :

On souhaite définir une nouvelle commande dans la console permettant de créer des utilisateurs. Pour ce faire, exécuter la commande suivante de la console :

php bin/console make:command

pour définir une nouvelle commande **add:user.** La commande *make:command* a créé une nouvelle classe ***AddUserCommand*** dans le dossier **src/command**. Remplacer cette classe par la classe *AddUserCommand* en ligne sur Madoc. Essayer de comprendre cette classe et utiliser la pour créer un nouveau administrateur de l'application.