TP2 – Symfony : Doctrine & Forms

I. Préparation

- Nous allons développer un ***CRUD*** pour ***l'inscription à une Newsletter***.
- Créer un nouveau Controller appelé ***InscriController***.

```
>composer require symfony/maker-bundle --dev
>php bin/console make : controller InscriController
```

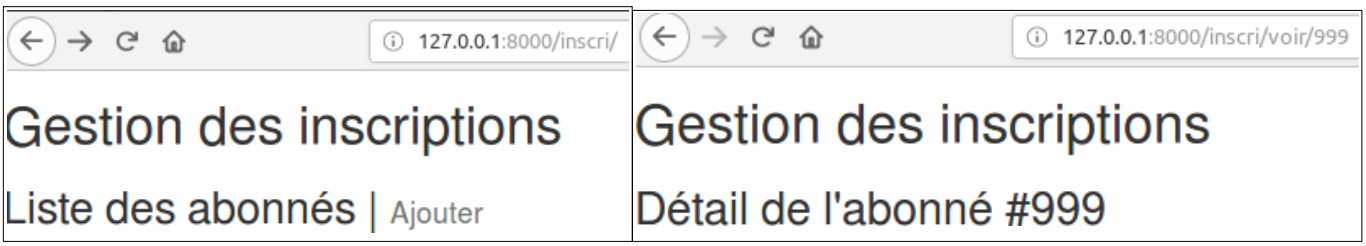
- Ajoutez les actions suivantes en tenant en compte la route et son nom :

Route	*Nom de la route*	*Action*
/inscri/	inscri_accueil	index
/inscri/voir/{id}	inscri_voir	voir
/inscri/ajouter/	inscri_ajouter	ajouter
/inscri/supprimer/{id}	inscri_supprimer	supprimer

- Créez un layout de base **templates/inscri-layout.html.twig**

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>
{% block title %}
TP Symfony #2 : Doctrine &Forms
{% endblock %}
</title>
<link rel="stylesheet" type="text/css" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css ">
</head>
<body>
<div class="container">
<h1>Gestion des inscriptions</h1>
{% block body %}
{% endblock %}
</div>
</body>
</html>
```

- Créer pour chacune des actions du contrôleur - un fichier Twig (template) dont le nom est celui de l'action(exp : ***/inscri/index.html.twig*** pour l'action **index**)
- Chaque fichier ***hérite*** d'***inscri-layout.html.twig*** et doit ***redéfinir*** le bloc «body» en affichant une balise <h2> avec les libellés suivants:
  - «Liste des abonnés|<small>Ajouter</small>» pour le template de l'action index
  - «Détail de l'abonné#x» le template de l'action voir, x étant l'id passé en paramètre
  - «Ajouter un abonné» pour le template de l'action ajouté
  - Pas de twig pour la suppression.
- Modifier les méthodes du contrôleur afin qu'elles retournent la page adéquate.
- **Testez** pour vérifier que toutes les URLs fonctionnent correctement :



II. Mise en œuvre de la couche modèle avec Doctrine

a) Installer Doctrine

- Installez le support Doctrine via l'***orm pack Symfony***
- Installez le ***MakerBundle***, qui vous aidera à générer du code

```
>composer require symfony/orm-pack
>composer require --dev symfony/maker-bundle
```

b) Création de la base de données

- Avant toute chose, il faut définir les paramètres de configuration pour Doctrine.
- Ouvrez le fichier ***.env*** et modifiez les paramètres nécessaires à Doctrine (en couleur de fond)

```
###> doctrine/doctrine-bundle ###
# Format described at http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a-url
# For an SQLite database, use: "sqlite:///%kernel.project_dir%/var/data.db"
# Configure your db driver and server_version in config/packages/doctrine.yaml
DATABASE_URL=mysql://root: @127.0.0.1:3306/inscri_VOTRE_NOM
###< doctrine/doctrine-bundle ###
```

- Maintenant que vos paramètres de connexion sont configurés, Doctrine peut créer la base de données ***inscri_VOTRE_NOM*** pour vous :

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

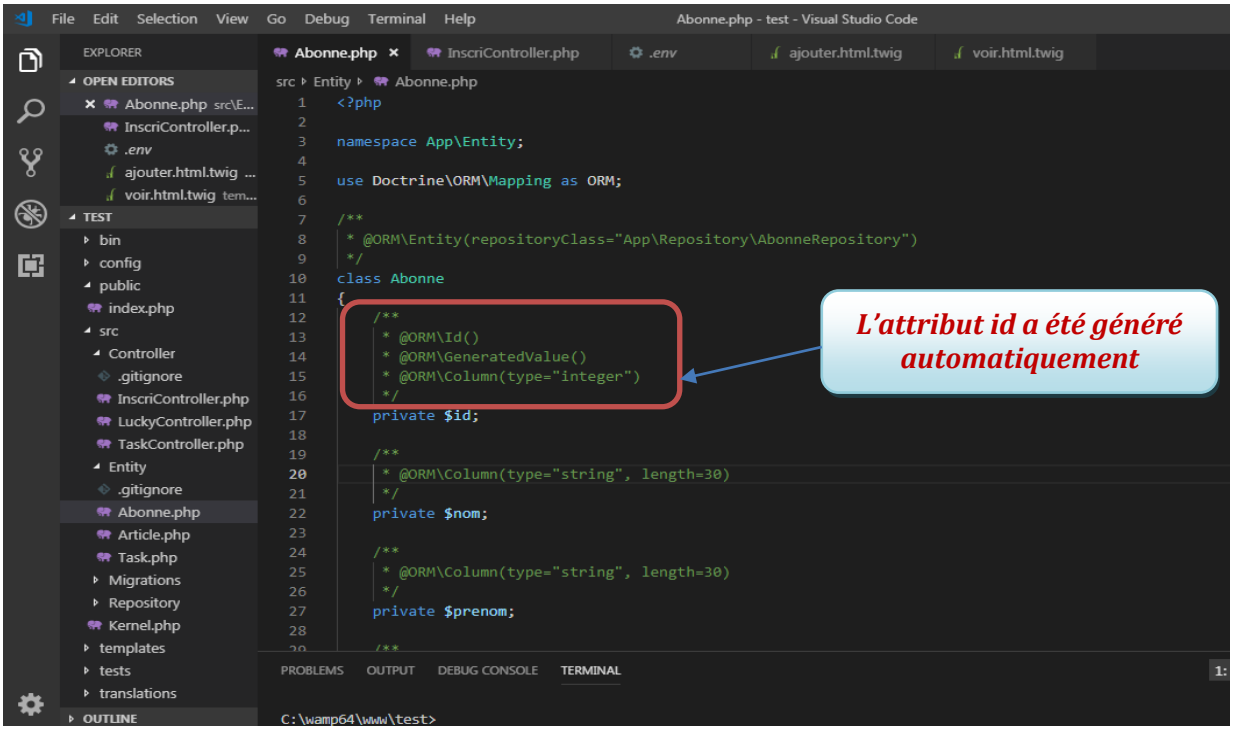
c) Création d'une table

- Supposons que vous construisez une application où les produits doivent être affichés.
- Sans même penser à Doctrine ou aux bases de données, vous savez déjà qu'il vous faut un Abonne objet pour représenter ces abonnés.
- Vous pouvez utiliser la ***make:entity*** commande pour créer cette classe et tous les champs dont vous avez besoin. La commande vous posera quelques questions - répondez comme ci-dessous :

```
>php bin/console make:entity

Class name of the entity to create or update:
> Abonne
New property name (press <return>to stop adding fields):
>nom
Field type (enter ? to see all types) [string]:
>string
Field length [255]:
>30
Can this field be null in the database (nullable) (yes/no) [no]:
>no
New property name (press <return> to stop adding fields):
```

>prenom
Field type (enter ? to see all types) [string]:
>string
Field length [255]:
> 30
Can this field be null in the database (nullable) (yes/no) [no]:
>no
New property name (press <return> to stop adding fields):
>dateNaissance
Field type (enter ? to see all types) [string]:
>date
Can this field be null in the database (nullable) (yes/no) [no]:
>yes
New property name (press <return> to stop adding fields):
>
(***press enter again to finish***) ***appuyez à nouveau sur enter pour terminer***

- Deux nouveaux fichiers ont été **créés** :
  - ***src/Entity/Abonne.php***
  - ***src/Repository/AbonneRepository.php***
- Un ***repository*** centralise tout ce qui touche à la récupération de vos entités.
- Vous ne devez pas faire la moindre requête SQL ailleurs que dans un repository, c'est la règle
- Doctrine a généré la classe ***Abonne*** (***Entity/Abonne.php***) qui vous servira dans votre application pour manipuler les abonnés.
- Nous avons choisi «***annotations***» comme format de ***mapping***, vous pouvez constater que votre classe et ses attributs ont été «***décorés***» par des commentaires comportant le terme «***ORM***» : il s'agit des annotations.
- Elles seront utilisées par Doctrine pour réaliser le ***mapping*** objet-relationnel.



- Pour chacun des attributs, Doctrine a également généré des getters/setters

sauf l'id pour lequel il a généré uniquement un getter (et PAS de setter!!)

- Une fois l'entité (la classe créée), nous allons demander à Doctrine de créer la table sous MySQL. En mode console saisir la commande suivante:

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

- Si vous ouvrez ce fichier, il contient le code SQL nécessaire pour mettre à jour votre base de données ! Pour exécuter ce SQL, exécutez vos migrations :

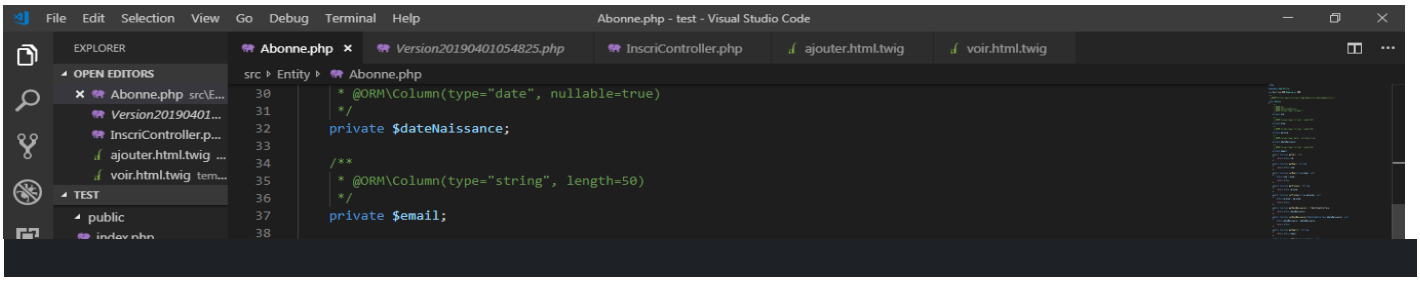
```
>php bin/console doctrine:migrations:migrate
```

- Cette commande exécute tous les fichiers de migration qui n'ont pas encore été exécutés sur votre base de données.
- Vous devez exécuter cette commande sur la production lors du déploiement pour maintenir votre base de données de production à jour.

a) Ajout d'un attribut après création de labase

- On voudrait ajouter un champ ***email*** de l'abonné.
- Il faut modifier la classe **Abonne** pour y ajouter le nouvel attribut (ou le modifier ou le supprimer).
- Ajouter ceci à la classe (entity) ***Abonne.php***, juste après l'entité «dateNaissance» :

> php bin/console make:entity
Class name of the entity to create or update:
> Abonne
New property name (press <return> to stop adding fields):
>email
Field type (enter ? to see all types) [string]:
>string
Field length [255]:
> 50
Can this field be null in the database (nullable) (yes/no) [no]:
>no
New property name (press <return> to stop adding fields):
>
(***press enter again to finish***) ***appuyez à nouveau sur enter pour terminer***



- La nouvelle propriété est mappée, mais elle n'existe pas encore dans la table ***abonne***. Aucun problème ! Générez une nouvelle migration :

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

➔Le SQL dans le fichier généré ressemblera à ceci :

```
ALTER TABLE abonne ADD email VARCHAR(50) NOTNULL
```

- Le système de migration est *intelligent*. Il compare toutes vos entités avec l'état actuel de la base de données et génère le code SQL nécessaire pour les synchroniser ! Comme auparavant, exécutez vos migrations :

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

III. Gestion du contrôleur et des vues

Nous allons dans cette partie, traiter les différentes actions du contrôleur les unes après les autres:

a) Action index

La page retournée par cette action doit afficher la liste des abonnés.

❖ **Modification du contrôleur**

- On doit indiquer l'utilisation de la classe Abonne. Ajoutez l'instruction suivante:

```
use App\Entity\Abonne;
```

- Modifiez la méthode index comme ci-après

```
public function index(){
$em = $this->getDoctrine()->getManager();
$repo = $em->getRepository(Abonne::class);
$lesAbonnes = $repo->findAll();
return $this->render('inscri/index.html.twig', ['lesAbonnes' =>$lesAbonnes]);
}
```

- ***Explications :*** La première partie du code peut être décomposée ainsi :

```
// On appelle le serviceDoctrine
$doctrine =$this->getDoctrine();
// On récupère un objet EntityManager responsable du processus
// de persistance et de la récupération des objets
$em= $doctrine->getManager();
// Grâce à cet objet EntityManager, on crée un dépôt de tous nos abonnées
$repository = $em->getRepository(Abonne::class);
// On récupère tous les abonnés du dépôt
$lesAbonne= $repository->findAll();
// QUESTION : où est définie la méthode findAll() ?
```

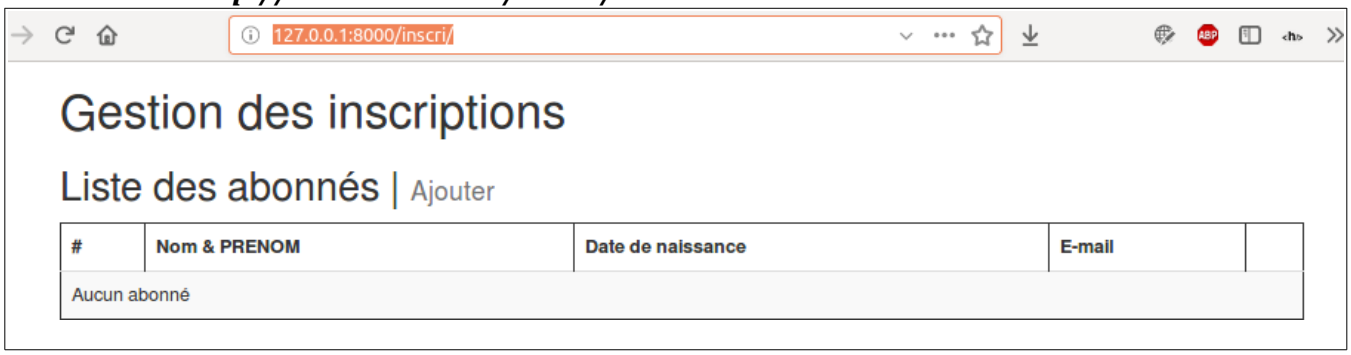
❖ **Modification de la vue (**templates/inscri/index.html.twig**)**

Voici le nouveau code de la vue (à compléter)

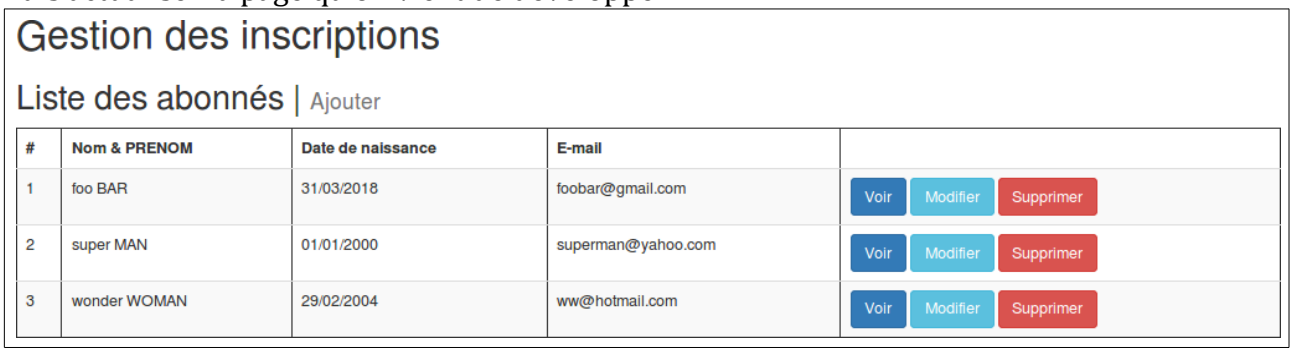
```
{% extends ... %}
{% block ... %}
<h2>Liste des abonnés | <small>Ajouter</small></h2>
<table class="table table-hover table-striped" border="1">
<thead>
<tr>
<th>#</th>
<th>Nom & Prénom</th>
<th>Date de naissance</th>
<th>E-mail</th>
<th></th>
</tr>
</thead>
<tbody>
{% for abonne in ... %}
<tr>
<td>{{abonne.id}}</td>
<td>{{abonne....}} {{abonne.... | upper}}</td>
<td>{{abonne.dateNaissance | ...('d/m/Y')}}</td>
<td>{{abonne.    }}</td>
<td>
```

```
<a class="btn btn-primary" href="#">Voir</a>
<a class="btn btn-primary" href="#">Voir</a>
<a class="btn btn-info" href="#">Modifier</a>
<a class="btn btn-danger" href="#">Supprimer</a>
</tr>
</td>
{% else %}
<tr>
<td colspan="5">Aucun abonné</td>
</tr>
{% endfor %}
</tbody>
</table>
{% endblock %}
```

Ouvrir l'url ***http://127.0.0.1:8000/inscri/.*** On devrait obtenir ceci :



En utilisant ***phpMyAdmin*** ou autre, insérer manuellement 3 enregistrements dans la table *abonne*. Puis actualiser la page qu'on vient de développer :



b) Action « voir»

❖ ***Modification du contrôleur***

- C'est presque la même chose mais on appelle cette fois-ci la méthode ***find()*** du repository au lieu de la méthode ***findAll().***
- On vérifie l'existence de l'abonné. S'il n'existe pas on lance une exception, sinon on passe à la vue un abonné et non plus un tableau d'abonnés.
- Modifier la méthode voir comme suit (à compléter) :

```
public function voir($id)
{
$em = $this->getDoctrine()->getManager();
$repo=$em->getRepository(___);
// On récupère les données de l'abonné
$abonne=$repo->_____($_____);
// S'il n'exite pas on lance une exception (erreur 404)
if($abonne == null)
throw$this->createNotFoundException("404- Abonné $id inéxistant !");
// On le passe à la vue pour l'afficher
return $this->render('inscri/voir.html.twig', ["abonne"=>$_____]);
}
```

❖ ***Modification de la vue(voir.html.twig)***

- Faites les modifications nécessaires pour obtenir l'affichage suivant (attention à la date)

Testez en utilisant un identifiant existant.

- Testez également le cas d'une requête comportant un id inconnu, par exemple : http://127.0.0.1:8000/inscri/voir/55
  ➔ On est dans l'environnement de ***DEV*** (app_dev.php), symfony affiche les détails de l'erreur.
- Testez maintenant la même URL dans l'environnement de prod : http://localhost:8000**/index.php/**inscri/voir/55
  ➔ Qui affiche une page différente sans aucun détail technique.
- On va personnaliser cette page en lui appliquant le layout de base : créez alors le fichier suivant (ainsi que les dossiers manquant) ***/templates/Exception/error404.html.twig*** y insérer le code suivant :

```
{% extends 'inscri-layout.html.twig' %}
{% block title %}{{parent()}} | Erreur 404{% endblock %}
{% block body %}
<h2>Oops : La page que vous cherchez n'existe pas !!</h2>
{% endblock %}
```

- Sauvegardez le fichier et testez de nouveau : il se peut que rien ne change !! C'est normal car on est en mode production, il faut donc vider le cache du prod manuellement :

```
>php bin/console cache:clear --env=prod
```

- Testez de nouveau et assurez-vous que le message d'erreur utilise bien la charge graphique globale.

c) Action ajouter

Cette partie se fera en deux étapes : ***créer le formulaire*** ensuite ***traiter sa validation***.

- ***Modification du contrôleur pour générer le formulaire de saisie***

- Il est nécessaire de disposer d'un formulaire de saisie.
- Pour Symfony, un formulaire se construit sur un ***objet existant***, et son objectif est ***d'hydrater cet objet.***
- **Hydrater**, dans le jargon Symfony, signifie que le formulaire va remplir les attributs de l'objet avec les valeurs entrées par l'utilisateur.
- Ce n'est qu'une fois l'objet hydraté que vous pourrez en faire ce que vous voudrez : enregistrer en base de données dans le cas de notre objet Abonne.
- Le système de formulaire ne s'occupe pas de ce que vous faites de votre objet, il ne fait que l'***hydrater***.
- Concrètement, pour créer un formulaire, on a besoin:
  - ✓ d'un ***objet*** (ici, notre objet Abonne).
  - ✓ d'un «***constructeur de formulaire***» (un FormBuilder)
- Il faut alors préciser au FormBuilder :
  - ✓ quels attributs de l'objet doivent être pris en compte par le formulaire
  - ✓ sous quelle forme ils doivent apparaître (champ texte, date, case à cocher,…)
  - ✓ On peut également définir pour chaque champ un tableau d'options : label, taille du champ texte, champ non obligatoire, sélection multiple, …
- A l'aide du FormBuilder on va alors générer le formulaire :

```
public function ajouter()
{
$abonne = new Abonne();
$fb = $this->createFormBuilder($abonne)
->add('nom', TextType::class)
->add('prenom', TextType::class, array("label" =>"Prénom"))
->add('dateNaissance', DateType::class)
->add('Valider', SubmitType::class);
// générer le formulaire à partir du FormBuilder
$form = $fb->getForm();
// Utiliser la méthode createView() pour que l'objet soit exploitable par la vue
return$this->render('inscri/ajouter.html.twig', ['f' => $form->createView()] );
}
```

- Les types des champs (TextType...) doivent être importés via l'instruction «***use***», les classes sont définies dans le namespace suivant :
  - ***Symfony\Component\Form\Extension\Core\Type\???***
- On génère un formulaire qui contient tous les attributs de l'objet sauf l'id.

❖ ***Modification de la vue(ajouter.html.twig)***

Juste 3 petites lignes à ajouter :

```
5
6  {% block body %}
7      <h2> Ajouter un abonné </h2>
8      {{ form_start(f) }}
9          {{ form_widget(f) }}
10     {{ form_end(f) }}
11 {% endblock %}
```

- ✓ **form_start()**- Affiche la balise d'ouverture <form>
- ✓ **form_widget()**- Affiche les champs du formulaire et leur label.
- ✓ **form_end()** - Affiche la balise de fermeture du formulaire ainsi que tous les champs qui n'ont pas encore été affichés.

➔Et voilà le résultat !

- On remarque qu'il manque le champ «Email», compléter le code en utilisant le **bon format**: symfony.com/doc/current/reference/forms/types.html
- Tester le formulaire : la validation ne fait rien (les données ne sont pas sauvegardées dans la base!!) Ci-dessous comment résoudre le problème:

❖ **Modification du contrôleur pour le traitement de la réponse**

- Par défaut, le formulaire envoie une requête POST au contrôleur qui l'a généré
- Maintenant on doit ***injecter*** les données saisies via le formulaire dans les propriétés d'un objet : ces données sont stockées dans le paramètre ***$request*** passé à l'action courante, ajoutez donc un nouveau paramètre à la méthode :

```
public function ajouter(Request $request)
```

- Sachant que la classe ***Request*** est déclarée dans le namespace suivant :

```
Symfony\Component\HttpFoundation\Request
```

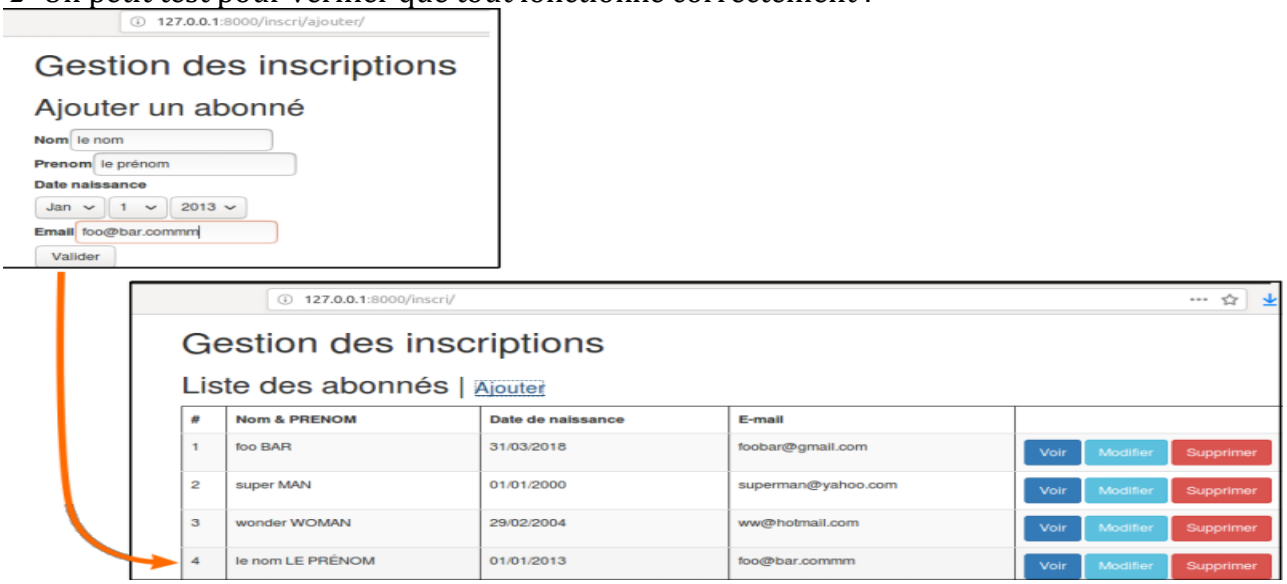
- Complétez votre code comme ci-après (partie en **gras**). N'hésitez pas à poser des questions si cela vous semble obscur.

```
/**
* @Route("/inscri/ajouter/", name="inscri_ajouter")
*/
public function ajouter()
{
$abonne = new Abonne();
$form = $this->createFormBuilder($abonne)
->add('nom', TextType::class)
->add('prenom', TextType::class)
->add('dateNaissance', DateType::class)
->add('email', EmailType::class)
->add('Valider', SubmitType::class)
->getForm();

$form->handleRequest($request);

if($form->isSubmitted()) {
$em = $form ->  ;
$em=$this->getDoctrine()->getManager();

$em->persist($abonne);
$em->flush();

return $this->redirectToRoute('inscri_accueil');
}
return $this->render('inscri/ajouter.html.twig', ['f' => $form->createView()]);
}
```

Notre contrôleur propose maintenant 3 comportements possibles :

- Au chargement initial de la page, le formulaire est créé et affiché.
- La méthode ***handleRequest ()*** détecte que le formulaire n'a pas été envoyé et ne fait rien.
- La méthode ***isSubmitted()*** renvoie false
- Quand l'utilisateur envoie le formulaire, la méthode ***handleRequest ()*** détecte l'événement «***submit***» et hydrate l'objet «***abonne***» avec les données du formulaire.
- Enfin on demande à doctrine de faire ***persister l'objet hydraté***.

➔ Un petit test pour vérifier que tout fonctionne correctement :



➔ On teste également un cas d'erreur :

❖ Ajoutez un message

- On va améliorer un peu notre interface, et afficher un message indiquant que l'abonné a bien été ajouté.
- Nous allons utiliser ***les messages flash***. Il s'agit en fait d'une ***variable de session*** qui ne dure que le temps d'une seule page.
- La page qui traite le formulaire définit un message flash «***Abonné bien ajouté***» puis redirige vers la liste de l'abonnés. Sur cette page, le message flash s'affiche puis est détruit de la session. Ainsi, si on actualise de page le message flash ne sera plus présent.
- Modifiez le code de la méthode ajouter () comme ci-dessous (à compléter) :

```
...
$em->flush();
$session = new Session();
$session->getFlashBag()->add('notice', 'Abonnée ...');
return $this→redirectToRoute('inscri_accueil');
...
```

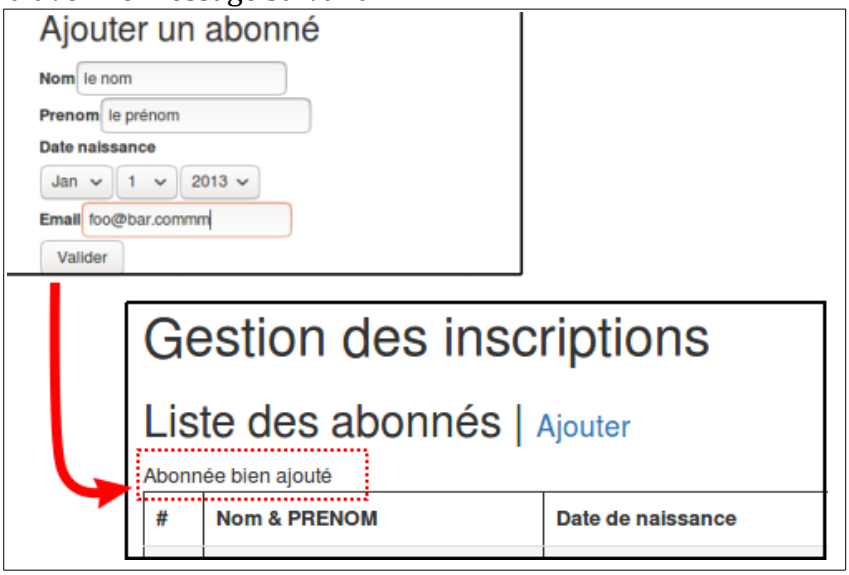
- la classe «***Session***» est définie dans le namespace suivant :

```
Symfony\Component\HttpFoundation\Session\Session
```

- Modifiez le code de la vue ***index.html.twig*** comme ci-dessous :

```
    {% block body %}
<h2>Liste des abonnés | <small><a href="{{path('inscri_ajouter')}}">Ajouter</a></small></h2>
    {% for msg in app.flashes('notice') %}
    {{msg}}
    {% endfor %}
    <table class="table table-hover table-striped" border="1">
    ...
```

- Testez et vérifier d'avoir le message suivant :



- si vous actualisez la page actuelle (***F5***), le message ne s'affiche plus ! C'est l'intérêt d'un message «***flash***» !
- On va mettre en évidence le message flash, modifier le twig comme ceci :

```
{% for msg in app.flashes('notice') %}
<div class="alert alert-success">
{{msg}}
</div>
{% endfor %}
```

❖ ***Format de la date dans le formulaire***

- Par défaut Symfony propose les dates la forme de 3 listes déroulantes. De plus on est limité à 10 années.
- Pour conserver les listes déroulantes, mais en ayant plus d'années disponibles : Il faut utiliser la classe ***BirthdayType*** au lieu de DateType (importer le namespace adéquat)

```
...
$form = $this->createFormBuilder($abonne)
...
->add('dateNaissance', BirthdayType::class)
...
```

- Faire un test pour vérifier les modifications.
- Pour obtenir un champ de type «text», il faut ajouter une option dans le tableau des options:

```
$form = $this->createFormBuilder($abonne)
->add...
->add('dateNaissance', DateType::class, ["widget" => "single_text"])
->...
```

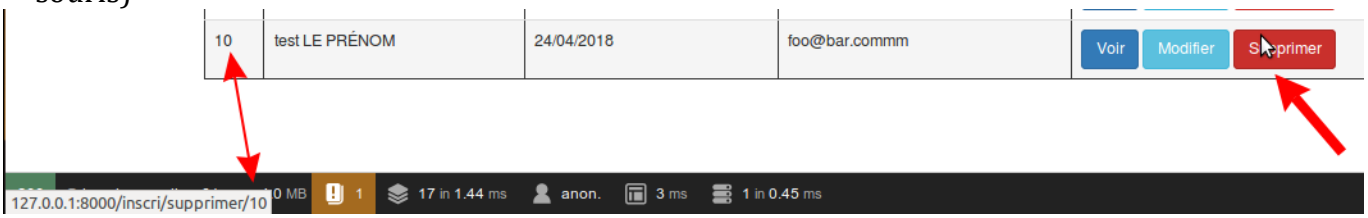
d) Action supprimer

❖ Cette action n'aura pas de vue, elle est accessible depuis le bouton «Supprimer» de la liste des abonnés.

❖ On commence alors pour mettre en place cette liaison. Modifier «***index.html.twig***» comme suit:

```
...
<td>
<a class="btn btn-primary" href="#">Voir</a>
<a class="btn btn-info" href="#">Modifier</a>
<a class="btn btn-danger"
href="{{path('inscri_supprimer', {'id':abonne.id})}}">Supprimer</a>
</td>
</tr>
...
```

❖ Vérifier d'avoir le même résultat que ceci (ATT : ne pas cliquer, juste pointer avec la souris)



❖ Compléter le traitement de suppression :

```
/**
* @Route("/inscri/supprimer/{id}", name="inscri_supprimer")
*/
public function supprimer($id)
{
$em = ...;
$item = $em->getRepository(...)->find(...);
$em->remove(...);
$em->flush();
$session = ...;
$session->getFlashBag()->add('notice', "L'abonnée $id a ...");
return $this->redirectToRoute(...);
}
```

❖ Vérifier que la suppression est fonctionnelle.

- Il y a un problème : la suppression se fait sans confirmation, il faut donc demander à l'utilisateur de valider cette action : Ajouter celle ligne à ***inscri-layout.html.twig*** :

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
```

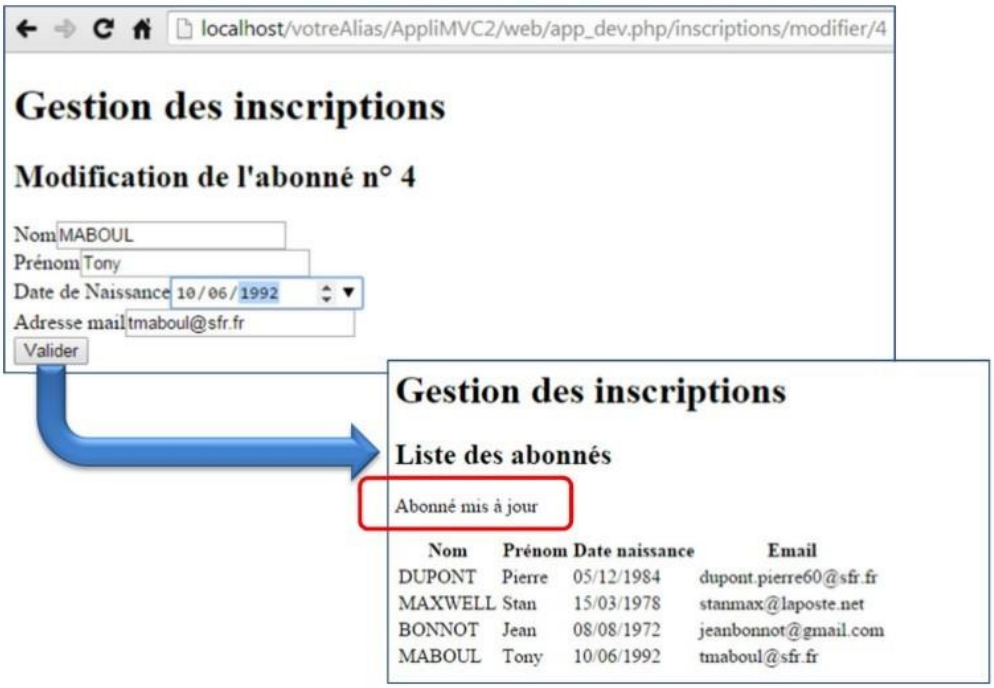
- Ensuite ajouter à la fin ***d'index.html.twig*** le code suivant :

```
...<script>
$(".btn-danger").click(function(){
returnconfirm("Supprimer ?");
})
</script>
{% endblock %}
```

e) Action modifier

- On veut pouvoir modifier un abonné via l'url suivante : http://127.0.0.1:8000/inscri/modifier/5 (5 est juste un exp, ça peut être l'identifiant de n'importe quel abonnée)
- Créer la méthode modifier () en prenant en compte ceci :
  - ***Route :*** à vous de trouver
  - ***Nom de la route :*** inscri_modifier
- Le code de cette méthode est identique à celui de la ***méthode ajouter ()*** avec quelques petites adaptations :
  - ***$abonne = new Abonne ();***et la remplacer par les instructions permettant de récupérer les informations de l'abonné dont l'***id*** est passé en paramètre (aidez-vous de la méthode voir()).
  - ***Changer le nom de la vue*** dans l'appel de la méthode ***render*** (et éventuellement créer la vue qui est une copie de la vue d'ajout).
  - Passer le formulaire il faut également passer l'abonné!
  - On redirige ensuite vers la page contenant la liste de tous les abonnés en affichant un message de confirmation

➔Vous devez obtenir ceci :



- Une fois le formulaire de modification fonctionnel, modifier ***index.html.twig*** afin que le bouton «Modifier» soit lié à ce formulaire.

```
<a class="btn btn-info"href="{{path('inscri_modifier', {'id':abonne.id})}}">Modifier</a>
```

```
<script>
$(".btn-info").click(function(){
Return confirm("Modifier ?");
})
</script>
```

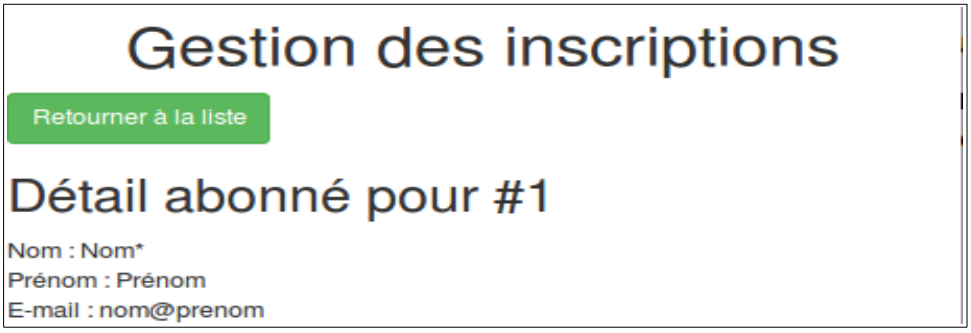
IV. Améliorations des IHM et de la navigation

On voudrait ajouter un lien-bouton dans toutes les pages (sauf l'index) pour revenir à la liste (index).On a deux solutions:

1) Ajouter ce lien dans les 3 vues: ajouter, modifier, voir

2) Ajouter ce lien dans ***inscri-layout*** et faire en sorte qu'il ne s'affiche pas dans la page d'accueil

Voici le résultat final :

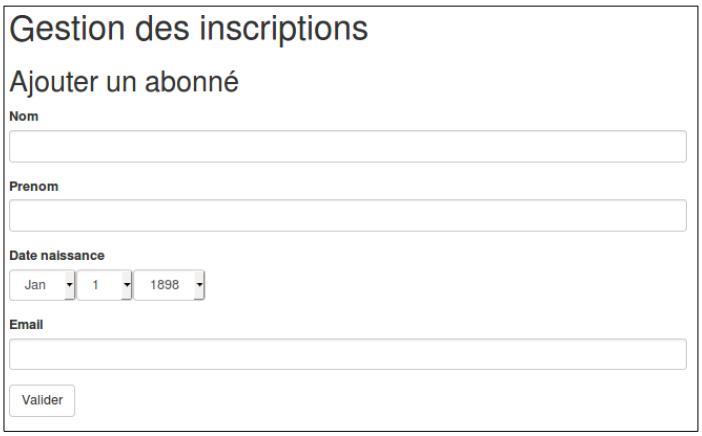


- On va mettre en forme le formulaire d'ajout qui n'est pas très beau (voir ci-contre) : les champs ne sont pas alignés, le bouton de validation n'est pas du même style que celui du haut, les lignes sont trop proches…

- Modifier le fichier ***config/package/twig.yaml*** en ajoutant la ligne en **gras**:

```
…
twig:
default_path: '%kernel.project_dir%/templates'
debug: '%kernel.debug%'
strict_variables: '%kernel.debug%'
form_themes: ['bootstrap_4_layout.html.twig']
…
```

- Vérifier que les formulaires d'ajout et de modification sont bien affichés comme ceci :



V. Simplification du code via les Param Converters

❖ Modifier l'action «***voir***» afin d'avoir le code simplifié suivant :

```
public function voir(Abonne $abonne)
{
return $this->render( 'inscri/voir.html.twig', ["abonne" => $abonne] );
}
```

❖ Tester cette action et vérifier que le résultat final n'a pas changé.
Qu'est ce qui s'est passé? Le paramètre {id} a été automatiquement convertir en objet Abonné grâce au bundle «***ParamConverter***» : Les ***ParamConverters*** ou «***convertisseurs de paramètres***» permettent de convertir des paramètres de requêtes en objets.

❖ Ces objets sont stockés comme attributs de requête de telle sorte qu'ils puissent être ***injectés*** comme arguments de méthodes de contrôleur.
Appliquer la même méthode pour toute action qui prend l'identifiant en paramètre et tester que tout fonctionne bien.

VI. Réduire le code des formulaires

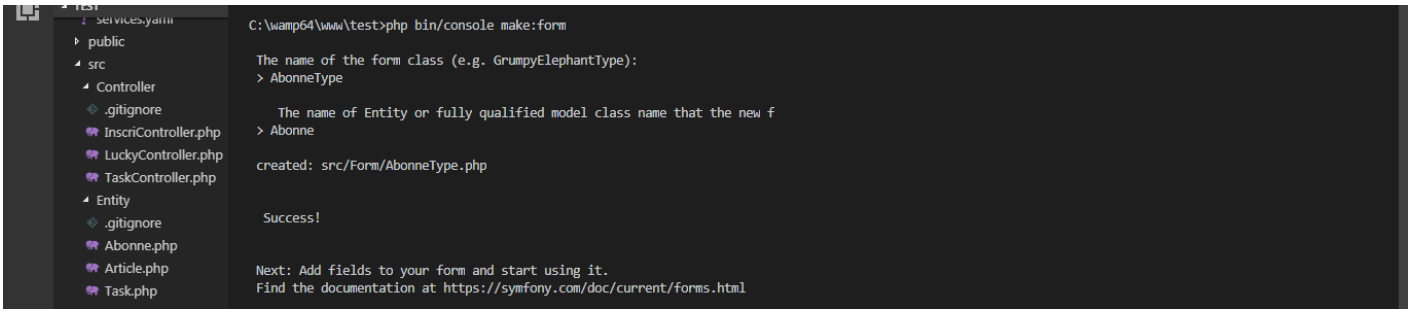
❖ On remarque que ajouter () et modifier () sont très similaires et il ya un code qui se répète dans les 2 méthodes : la création du formulaire.

❖ Pour simplifier suivre les étapes suivantes:

❖ Créer un formulaire :

php bin/console make :form

❖ Voici le contenu du fichier «***AbonneType.php***» :



```
<?php
namespaceApp\Form;
use App\Entity\Abonne;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
class AbonneType extends AbstractType
{
public function buildForm(FormBuilderInterface $builder, array $options)
 {
$builder
        ->add('nom')
        ->add('prenom')
        ->add('dateNaissance')
        ->add('email')
    ;
  }
public function configureOptions(OptionsResolver $resolver)
  {
$resolver->setDefaults(['data_class' =>Abonne::class,]);
  }
}
```

- Quelques remarques concernant le nouveau fichier :
  - il appartient au namesapce «***App\Form***»
  - Il permet de définir uniquement la structure du formulaire via le paramètre«***$builder***» de la méthode «***buildForm()***»
  - Il est relatif à l'entité «***App\Entity\Abonne***» (c'est la valeur du paramètre data_class)
- Modifier le code de la méthode ajouter () dont voici une partie du code :

```
56  {
57      $abonne = new Abonne();
58
59      $fb = $this->createFormBuilder($abonne)
60      ->add('nom', TextType::class)
61      ->add('prenom', TextType::class, array("label" => "Prénom"))
62      ->add('email', EmailType::class)
63      ->add('dateNaissance', BirthdayType::class)
64
65      ->add('Valider', SubmitType::class);
66
67      // générer le formulaire à partir du FormBuilder
68      $form = $fb->getForm();
```

- Supprimer ce qui est encadré et le remplacer par ceci :

```
$form=$this->createForm(AbonneType::class,$abonne);
```

Insérer ce code:

```
use App\Form\AbonneType;
```

- Enfin déplacer les use relatifs aux classes Type (les types de champs) dans le nouveau fichier***AbonneType.php.***
- Tester et vérifier que le formulaire d'ajout fonctionne correctement.

Un peu de vocabulaire

Quand on appelle une constante, fonction ou classe, alors le nom de l'élément appelé est ***qualifié*** ou ***non qualifié***.

- ***qualifié*** si le namespace dans lequel il se trouve est spécifié.
  Exemple : B\fonction() et \A\B\fonction() sont tous les deux des éléments qualifiés.

- ***Non qualifié*** si le namespace dans lequel il se trouve n'est pas spécifié.
  Exemple : fonction () est un élément non qualifié.

Le même principe est à appliquer à la méthode modifier (), sauf que cette fois on peut simplifier: «***AbonneType***» est qualifié. On peut la remplacer par ceci :

```
$form=$this->createForm(AbonneType::class,$abonne);
```

La syntaxe «***UneClasse::class***» est appelée ***FQCN*** : Fully Qualified Calss Name. Voici une définition assez simple (et générale) : https://en.wikipedia.org/wiki/Fully_qualified_name
Du coupon peut simplifier la valeur de 'data_class' en utilisant ceci:

```
$resolver->setDefaults(array('data_class' =>Abonne::class
```

Cependant il faut préalablement ajouter ceci :

```
use CatalogueBundle\Entity\Produit;
```

VII. QueryBuilder

❖ On voudrait permettre à l'utilisateur de faire une recherche par nom, la requête qu'on devrait exécuter sera:

```
SELECT * FROM abonne WHERE nom LIKE '%$critere%'
```

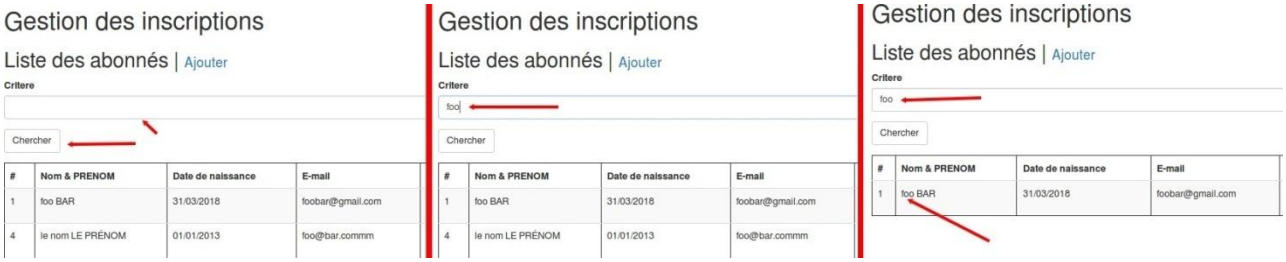
❖ Ajouter une nouvelle méthode dans le repository (à compléter)

```
class AbonneRepository extends ServiceEntityRepository
{
public function __construct(RegistryInterface $registry)
  {
     parent::__construct($registry, Abonne::class);
  }
public function recherche($data): array
{
   $entityManager = $this->getEntityManager();
   $query = $entityManager->createQuery(
      'SELECT a
      FROM App\Entity\Abonne a
     WHERE a.nom like :critere ')->setParameter('critere', $data['critere']);

   // returns an array of Product objects
return $query->execute();
}
}
```

❖ Modifier le controller «***index***» afin de créer un formulaire de recherche, détecte quand ce formulaire est validé, récupère le contenu du champ
❖ Enfin appelle la nouvelle fonction du ***repository***.
➔ Ci-dessous le code à compléter (à insérer juste avant le dernier *return*):

```
/**
   * @ROUTE("/inscri/",name="inscri_accueil")
*/
    public function index (Request $request,AbonneRepository $repo) {

// Créer un formulaire de recherche
$form = $this -> createFormBuilder ( )
        -> add ( 'critere' , TextType :: class,["required"=>false] )
        -> add ( 'chercher' , SubmitType :: class , [ 'label' => 'Chercher' ])
        ->getForm ();

     $form->($request);

if ($form->isSubmitted())
         {
        $data=->getData();

        $lesAbonnes=->recherche($data);

        }
else
     {
      $lesAbonnes=$repo->findAll();
  }
return $this -> render ( 'inscri/index.html.twig',['lesAbonnes'=>$lesAbonnes,'f'=>$form] );
}
```

- Modifier la **vue** afin d'afficher le formulaire juste **avant** le tableau. Tester et vérifier d'avoir un résultat proche de ceci:



- Problème : je fais une recherche, le filtre fonctionne bien, je veux supprimer le champ afin d'avoir la liste complète: impossible car le champ est obligatoire, le rendre donc non obligatoire:

```
...
->add("critere", TextType::class, ["required" => false])
```

VIII. Form Validation

- Le champ «nom» est obligatoire par défaut, on voudrait que sa taille soit comprise entre 3 et 15 caractères.
- Plusieurs solutions existent. La plus rapide étant de le faire dans l'entité: Editer le code de l'entité «Abonne» et ajouter ceci au début:

```
use Symfony\Component\Validator\Constraints as Assert;
```

- Ensuite modifier les annotations de l'attribut « nom » comme suit :

```
/**
    * @ORM\Column(name="nom",type="string", length=30)
* @Assert\Length(
    * min=3,
    * max=15,
    * minMessage="Minimum {{limit}} caractères",
    * maxMessage="Maximum {{limit}} caractères"
    * )
    */
   private $nom;
```

- Tester en ajoutant un abonné dont le nom contient **un seul caractère**: l'ajout fonctionne bien et on n'a aucun message d'erreur!!! Pour afficher un message il faut modifier la méthode ajouter () comme suit:

```
...
$form->handleRequest($request);
if($form->isSubmitted() && $form->isValid()) {
$em = $this->getDoctrine()->getManager();
...
```

- Tester de nouveau et vérifier d'avoir le résultat suivant :