Niveau : STS SIO 2ème année ☐ Cours ☐ TD ☑ TP

Module : SLAM5 – Solutions applicatives

Intitulé : Doctrine sous Symfony 2

Durée : 2 heures

Objectifs :
- ✓ Utiliser un ORM pour mettre en œuvre la couche modèle d'une application MVC
- ✓ Programmer au sein d'un Framework

Contenu

Objectif du TP

Vous allez dans ce TP réaliser une application web toute simple permettant de gérer des abonnés, à une newsletter par exemple.

L'application permettra d'afficher, d'ajouter, de modifier et de supprimer les abonnés. Cette application s'appuiera sur une architecture MVC.

La base de données ne contiendra qu'une seule table : la table Abonné.

Ce TP va vous permettre de mettre en application ce que vous avez appris lors du TP précédent et également de vous faire découvrir les ORM.

Ce sera l'occasion de vous faire découvrir la puissance des Frameworks MVC.

Travail préparatoire

- Créer un répertoire `AppliMVC2` dans votre dossier `www` sous OneDrive.
- Décompressez l'archive Symfony2 dans ce nouveau répertoire.
- Créez dans cette nouvelle application un bundle `Sio/InscriptionsBundle`
- Supprimez le bundle Acme (n'oubliez aucune étape !)
- Testez l'URL `/localhost/AppliMVC2/web/app_dev.php/hello/toto`
- Dans le fichier de routes de votre contrôleur, supprimez la route existante et ajoutez les routes suivantes :
  - `sioinscriptions_accueil` qui déclenchera l'action `index` du contrôleur `Inscriptions` de votre bundle et traitera les URL de type `/inscriptions/`
  - `sioinscriptions_voir` qui déclenchera l'action `voir` du contrôleur `Inscriptions` et traitera les URL de type `/inscriptions/inscription/{id}`
  - `sioinscriptions_ajouter` qui déclenchera l'action `ajouter` du contrôleur Inscriptions et traitera les URL de type `/inscriptions/ajouter/`
  - `sioinscriptions_supprimer` qui déclenchera l'action `supprimer` du contrôleur Inscriptions et traitera les URL de type `/inscriptions/supprimer/{id}`
- Renommez le contrôleur DefaultController en `InscriptionsController`. N'oubliez pas de modifier le nom de la classe !
- Créer les 4 méthodes permettant de gérer les 4 actions définies dans le fichier de routes. N'oubliez pas les paramètres de l'url !
- Créez un layout de base (`/app/Resources/views/layout.html.twig`) identique à celui du TP précédent. Vous modifierez simplement le titre de la page (TP MVC2) et le titre de niveau 1 (Gestion des inscriptions).

- Renommez le dossier Default de `/src/Sio/Inscriptions/Resources/views/` en `Inscriptions`.
- Créez dans ce dossier Inscriptions un template pour chacune des actions de votre contrôleur avec les caractéristiques suivantes, par rapport au layout de base :
  - bloc `title` inchangé
  - bloc `stylesheets` vide
  - bloc `javascipts` vide
  - bloc `body` : placez une balise `<h2>` avec les libellés suivants :
    - « Liste des abonnés » pour le template de l'action `index`
    - « Détail abonné pour » le template de l'action `voir`
    - « Ajouter un abonné » pour le template de l'action `ajouter`
    - « Supprimer un abonné » pour le template de l'action `supprimer`
- Modifier les méthodes du contrôleur afin qu'elles retourne la page adéquate.
- Testez pour vérifier que tout fonctionne correctement

Mise en œuvre de la couche modèle avec Doctrine

Vous allez voir maintenant l'intérêt d'un ORM comme Doctrine et comment il permet d'améliorer votre productivité !

Création de la base de données

➢ Avant toute chose, il faut définir les paramètres de configuration pour Doctrine.
Ouvrez le fichier `/app/config/parameters.yml`.
Modifiez les paramètres nécessaires à Doctrine. Ici, seul le nom de la base est à changer (tpmvc2).

```
parameters:
    database_driver: pdo_mysql
    database_host: 127.0.0.1
    database_port: null
    database_name: tpmvc2
    database_user: root
    database_password: null
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    mailer_user: null
    mailer_password: null
    locale: fr
    secret: ThisTokenIsNotSoSecretChangeIt
    debug_toolbar: true
    debug_redirects: false
    use_assetic_controller: true
```

- Afin d'éviter tout problème et pour s'assurer que votre modification sera bien prise en compte, vous allez vider le cache.
  Pour cela, ouvrez une console de commande sur le répertoire de votre application AppliMVC2 puis entrez : `php app/console cache:clear`

> Note
>
> Lorsque vous effectuez des changements dans votre application et que vous ne voyez pas les résultats escomptés, pensez à vider le cache !
>
> - En environnement de développement : `php app/console cache:clear`
> - En environnement de production : `php app/console cache:clear –env=prod`

- Allez, on y va pour la création de la base (vide, pour le moment).
  Toujours en mode console, tapez la commande suivante :

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

Vous pouvez vérifier sous PhpMyAdmin que la base de données tpmvc2 a bien été créée.

Création d'une table

- On va maintenant demander à Doctrine de créer notre table Abonné
  Voici la table que nous voulons créer :

Afficher | Structure | SQL | Rechercher

#	Nom	Type	Interclassement	Attributs
1	id	int(11)		
2	nom	varchar(30)	utf8_unicode_ci	
3	prenom	varchar(30)	utf8_unicode_ci	
4	dateNaissance	date		

En mode console, on saisit la commande permettant de créer une Entité (l'entité pour Doctrine sera votre classe sous Symfony et deviendra votre table sous MySQL) :

```
app/console doctrine:generate:entity
```

A partir de là, vous devez entrer un certain nombre d'informations :

- Le nom de l'entité
- Le type de fichier de mapping vous souhaitez obtenir (yaml, xml, php ou annotation)
- Pour chaque attribut, son nom, son type, éventuellement sa taille
- On vous propose ensuite de générer automatiquement le repository. Même si nous n'allons pas nous en servir dans ce tp, vous répondrez « oui » afin que nous examinions ce qui a été généré.

L'id ne doit pas être déclaré dans la liste des champs. Sa création est automatique et systématique.

Le type des attributs doit appartenir à la liste suivante : array, object, boolean, integer, smallint, bigint, string, text, datetime, datetimetz, date, time, decimal, float.

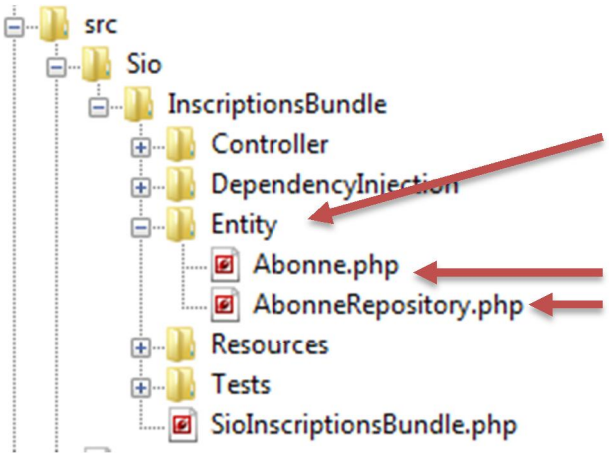
http://docs.doctrine-project.org/en/2.0.x/reference/basic-mapping.html

Allez, c'est parti :

```
C:\...\www\AppliMVC2> app/console doctrine:generate:entity

The Entity shortcut name: SioInscriptionsBundle:Abonne

Configuration format (yml, xml, php, or annotation) [annotation]:

New field name (press <return> to stop adding fields): nom
Field type [string]:
Field length [255]: 30

New field name (press <return> to stop adding fields): prenom
Field type [string]:
Field length [255]: 30

New field name (press <return> to stop adding fields): dateNaissance
Field type [string]: date

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

Do you want to generate an empty repository class [no]? yes
Do you confirm generation [yes]?
```

Attention au nom de l'entité : il faut la faire précéder du nom du bundle suivi de « : »

Voyons maintenant ce qu'il s'est passé du côté de l'application :



Un répertoire Entity a été créé dans votre bundle.
A l'intérieur de ce dossier, deux fichiers sont ajoutés :

- *Abonne.php (votre classe Abonne)*
- *AbonneRepository.php (le fameux repository dont on a demandé la génération automatique)*

- La classe Abonne (Entity/Abonne.php)

Doctrine a généré la classe Abonne qui vous servira dans votre application pour manipuler les abonnés.

Comme 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.

```php
<?php

namespace Sio\InscriptionsBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Abonne
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Sio\InscriptionsBundle\Entity\AbonneRepository")
 */
class Abonne
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string
     *
     * @ORM\Column(name="nom", type="string", length=30)
     */
    private $nom;

    /**
     * @var string
     *
     * @ORM\Column(name="prenom", type="string", length=30)
     */
    private $prenom;

    /**
     * @var \DateTime
     *
     * @ORM\Column(name="dateNaissance", type="date")
     */
    private $dateNaissance;

... / ...
```

L'attribut id a été généré automatiquement !

Pour chacun des attributs, Doctine a également généré des getters/setters (accesseurs en lecture et en écriture).

```php
    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set nom
     *
     * @param string $nom
     * @return Abonne
     */
    public function setNom($nom)
    {
        $this->nom = $nom;

        return $this;
    }

    /**
     * Get nom
     *
     * @return string
     */
    public function getNom()
    {
        return $this->nom;
    }

    /**
     * Set prenom
     *
     * @param string $prenom
     * @return Abonne
     */
    public function setPrenom($prenom)
    {
        $this->prenom = $prenom;

        return $this;
    }
```

[...]

- Le Repository (Entity/Abonne.php)

  Ce fichier est utilisé pour écrire vos propres requêtes d'accès aux données (DQL ou SQL QueryBuilder)

```php
namespace Sio\InscriptionsBundle\Entity;

use Doctrine\ORM\EntityRepository;

/**
 * AbonneRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class AbonneRepository extends EntityRepository
{
}
```

➢ Une fois la classe créée, nous allons demander à Doctrine de créer la table sous MySQL

En mode console, nous lançons une commande quie nous permet de vérifier ce que Doctrine faire faire :

```
app/console doctrine:schema:update --dump-sql
```

Cette commande nous retourne l'instruction SQL qui va être exécutée. Pensez à toujours lancer cette commande avant de lancer la création effective : c'est un moyen de contrôle qui permet de s'assuter que l'on a pas fait de bêtises.

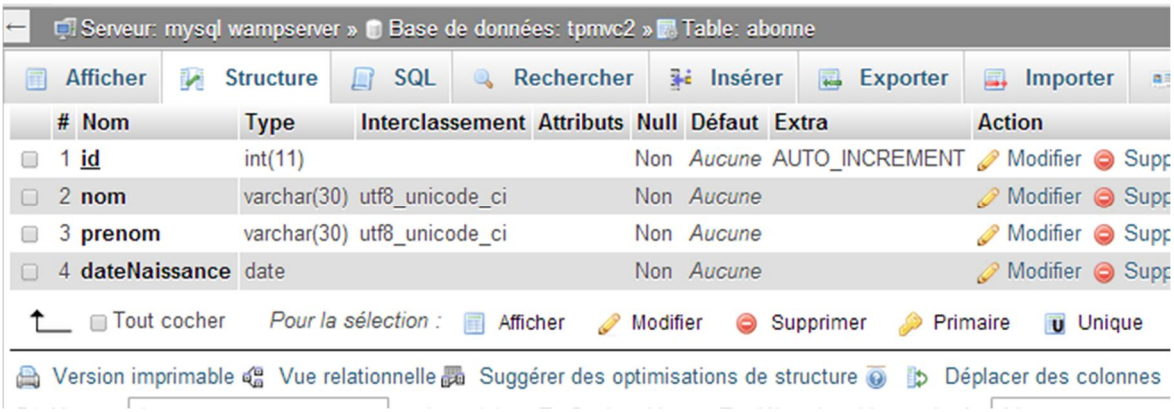
Vous obtenez :

```
CREATE TABLE Abonne (id INT AUTO_INCREMENT NOT NULL, nom
VARCHAR(30) NOT NULL, prenom VARCHAR(30) NOT NULL, dateNaissance
DATE NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE
utf8_unicod
e_ci ENGINE = InnoDB;
```

Constatant que c'est bien ce que l'on veut, on lance maintenant la création de la table :

```
app/console doctrine:schema:update --force
```

On vérifie sous PhpMyAdmin :



Fabuleux, n'est-ce- pas ?

Ajout d'un attribut après création de la base

L'erreur est humaine ! On peut donc s'être trompé ou avoir oublié quelque chose.

Disons, pour l'exemple, que l'on a oublié le champ email de l'abonné.

Pas de panique, Doctrine sait gérer.

Il suffit de modifier la classe Abonne pour y ajouter le nouvel attribut (ou le modifier ou le supprimer).

- Ajout le nouvel l'attribut dans l'entité

  Modifiez votre classe (Abonne.php) ainsi :

```
44
45      /**
46       * @var string
47       *
48       * @ORM\Column(name="email", type="string", length=50)
49       */
50      private $email;
51
```

- Ajout des getters/setters

  On pourrait également ajouter directement nos getters/setters mais à quoi bon : Doctrine peut le faire pour nous.

  Il suffit de lancer la commande suivante :

```
php app/console doctrine:generate:entities Sio/InscriptionsBundle/Entity/Abonne
```

Notez ici la manière de faire référence à notre Entité : on n'utilise pas le nom du bundle, comme lors de la création, mais le namespace complet.

Vérifiez dans votre classe la présence des getter/setter du nouvel attribut email.

- Mise à jour de la base de données

  Comme tout à l'heure, on vérifie d'abord :

```
app/console doctrine:schema:update --dump-sql
```

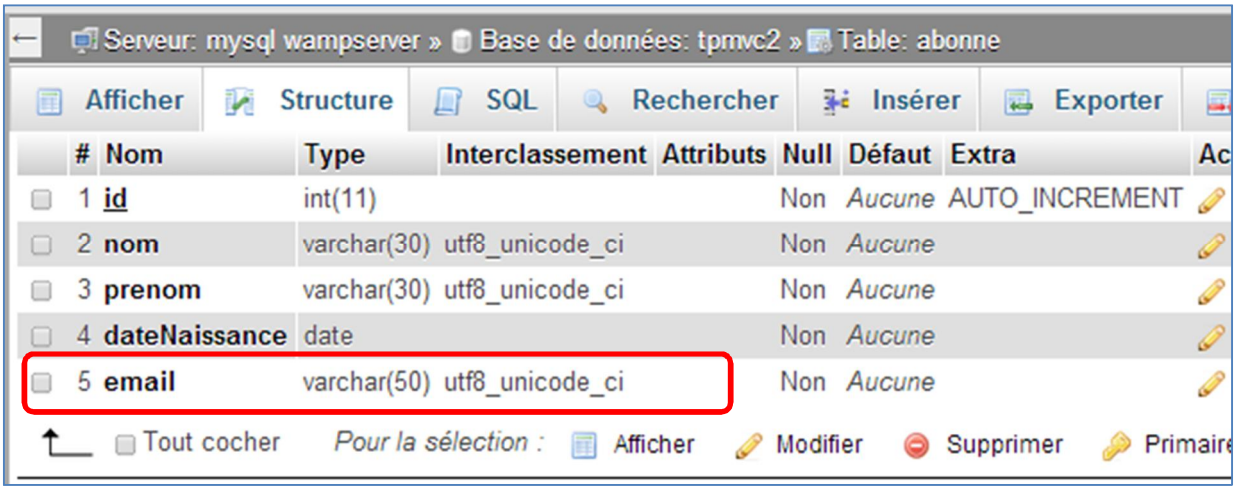
  Vous devez obtenir :

```
ALTER TABLE abonne ADD email VARCHAR(50) NOT NULL;
```

C'est correct, on lance donc la mise à jour de la table :

```
app/console doctrine:schema:update --force
```

Et pour finir, on vérifie sous PhpMyAdmin :



Et voilà, quelques commandes en mode console, et notre base de données et nos classes pour nos objets sont prêtes.

Finalement, avouez que la console a du bon …

Allez, il ne reste plus qu'à retourner dans notre contrôleur !

Gestion du contrôleur et des vues

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

Action index

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

- Modification du contrôleur

Il faut tout d'abord indiquer que l'on va utiliser notre classe Abonne :

Ajoutez donc l'instruction suivante :

```
use Sio\InscriptionsBundle\Entity\Abonne;
```

Modifiez la méthode indexAction comme ci-après :

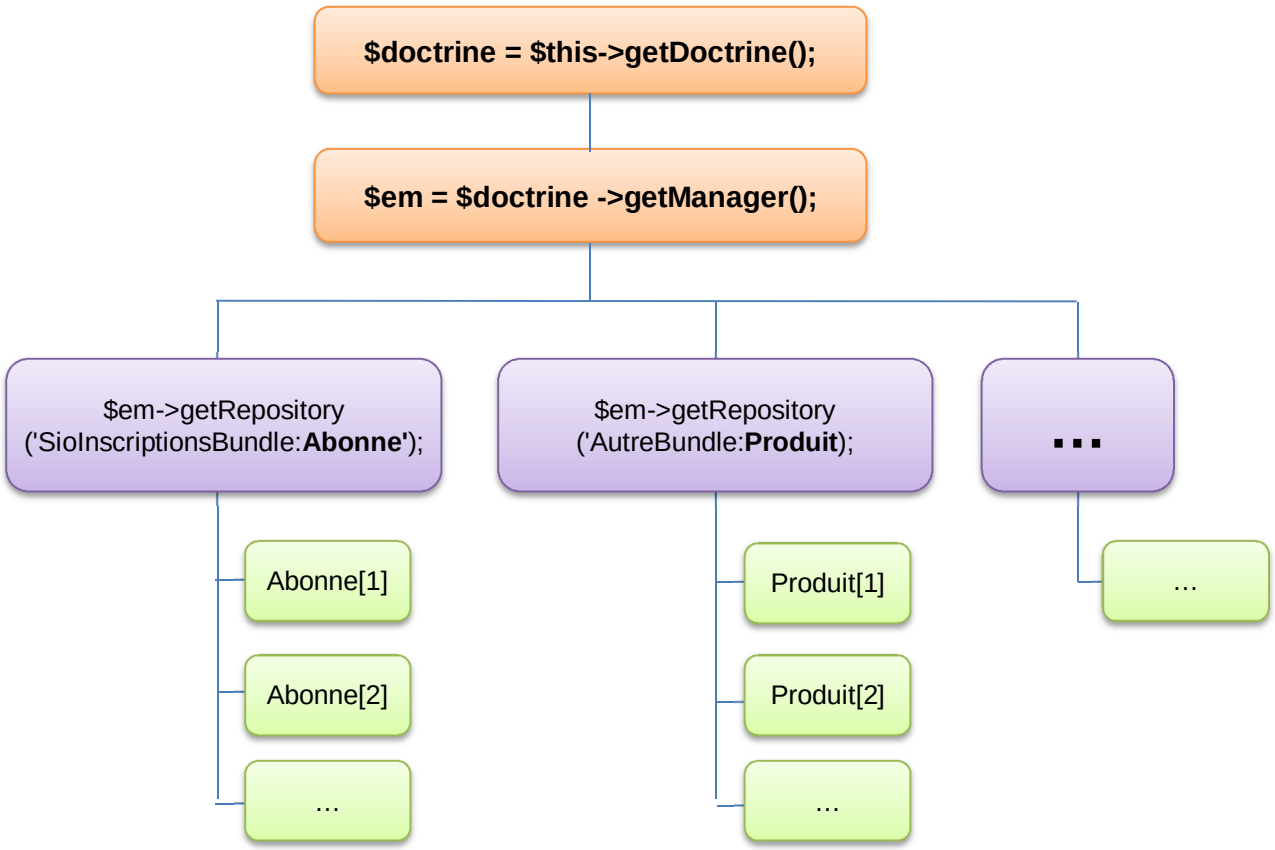
```php
class InscriptionsController extends Controller
{
    public function indexAction()
    {
        // On récupère le repository
        $repository = $this->getDoctrine()
                    ->getManager()
                    ->getRepository('SioInscriptionsBundle:Abonne');

        // On récupère tous les abonnés
        $lesAbonnes = $repository->findAll();
        return $this->render('SioInscriptionsBundle:Inscriptions:index.html.twig',
            array('lesAbonnes' => $lesAbonnes));
    }
```

Explications

- La première partie du code peut être décomposée ainsi :

```php
// On appelle le service Doctrine
$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('SioInscriptionsBundle:Abonne');

// On récupère tous les abonnés du dépôt
$abonne = $repository->findAll();
```

Un petit schéma récapitulatif :



- Demande à la vue de se mettre à jour :

```
// On demande à la vue de se mettre à jour
return $this->render('SioInscriptionsBundle:Inscriptions:index.html.twig',
    array('lesAbonnes' => $lesAbonnes));
```

On passe en paramètre à la vue le tableau des abonnés

➢ Modification de la vue (Resources/views/Inscriptions/index.html.twig)

```
{% extends "::layout.html.twig" %}

{% block stylesheets %}
{% endblock %}

{% block body %}
<h2>Liste des abonnés</h2>

<table>
    <thead>
        <tr>
            <th>Nom</th>
            <th>Prénom</th>
            <th>Date naissance</th>
            <th>Email</th>
        </tr>
    </thead>
    <tbody>
        {% for abonne in lesAbonnes %}
            <tr>
                <td>{{ abonne.nom }}</td>
                <td>{{ abonne.prenom }}</td>
                <td>{{ abonne.dateNaissance|date('d/m/Y') }}</td>
                <td>{{ abonne.email }}</td>
            </tr>
        {% else %}
            <tr>
                <td colspan="4">Aucun abonné !</td>
            </tr>
        {% endfor %}
    </tbody>
</table>
{% endblock %}

{% block javascripts %}
{% endblock %}
```

for ... else ...

Pas banal, cette affaire !!!

Lancez votre application. Vous devez obtenir :

Sous PhpMyAdmin, saisir 3 ou 4 enregistrements dans la table Abonne.

Puis relancez votre application.

Vous obtenez cette fois-ci :

Action voir

Modification du contrôleur

C'est quasiment 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 lève une exception.
On passe à la vue un abonné et non plus un tableau d'abonnés.

```
public function voirAction($id)
{
    // On récupère le repository
    $repository = $this->getDoctrine()
                ->getManager()
                ->getRepository('SioInscriptionsBundle:Abonne');

    // On récupère tous les abonnés
    $abonne = $repository->find($id);

    // Si aucun abonné n'a été trouvé avec l'id $id, on lève une exception
    if ($abonne == null)
    {
        throw $this->createNotFoundException("Abonné[id='$id] inexistant.");
    }

    return $this->render('SioInscriptionsBundle:Inscriptions:voir.html.twig',
        array('abonne' => $abonne));
}
```

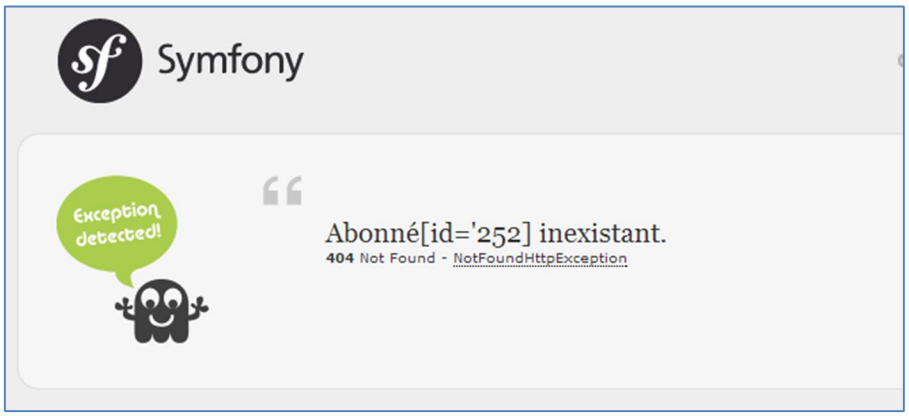
Modification de la vue (voir.html.twig)

Faites les modifications nécessaires pour obtenir l'affichage suivant :

Testez

Testez également le cas d'une requête comportant un id inconnu, par exemple : `http://localhost/votreAlias/AppliMVC2/web/app_dev.php/inscriptions/inscription/252`

Vous obtenez, en environnement de développement :



Et en environnement de production, la fameuse erreur 404 !

Action ajouter

➢ Modification du contrôleur pour générer le formulaire de saisie

Ici, il est nécessaire de disposer d'un formulaire de saisie.

Pour Symfony2, 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 Article, …. 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.

Allez, c'est parti :

```
public function ajouterAction()
{
    // On crée un objet Abonne
    $abonne = new Abonne();

    // On crée un FormBuilder et on lui ajoute les champs que l'on souhaite
    $formBuilder = $this->createFormBuilder($abonne)
        ->add('nom', 'text', array('label' => 'Nom'))
        ->add('prenom', 'text', array('label' => 'Prénom'))
        ->add('dateNaissance', 'date', array('label' => 'Date de Naissance'))
        ->add('email', 'email', array('label' => 'Adresse mail'))
        ->add('Valider', 'submit');

    // On génère le formulaire à partir du formBuilder
    $form = $formBuilder->getForm();

    // On passe la méthode createView() du formulaire à la vue afin qu'elle
    // puisse l'afficher
    return $this->render('SioInscriptionsBundle:Inscriptions:ajouter.html.twig',
        array('form' => $form->createView()));
}
```

Ici, 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 :

```
{% extends "::layout.html.twig" %}

{% block steelsheets %}
{% endblock %}

{% block body %}
    <h2>Ajouter un abonné</h2>
    {{ form_start(form) }}
        {{ form_widget(form) }}
    {{ form_end(form) }}
{% endblock %}

{% block javascripts %}
{% endblock %}
```

- form_start(form) - Affiche la balise d'ouverture <form>
- form_widget(form) - 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. C'est utile pour afficher les champs cachés et pour profiter de la protection CSRF automatique

Et voilà le résultat !

Bon, il n'est pas forcément très beau, mais avouez quand même que l'on n'a pas écrit beaucoup de lignes de code !

On verra un peu plus tard comment utiliser bootstrap pour nos CSS.

Maintenant on va s'intéresser au traitement de la validation du formulaire.

➢ Modification du contrôleur pour le traitement de la réponse

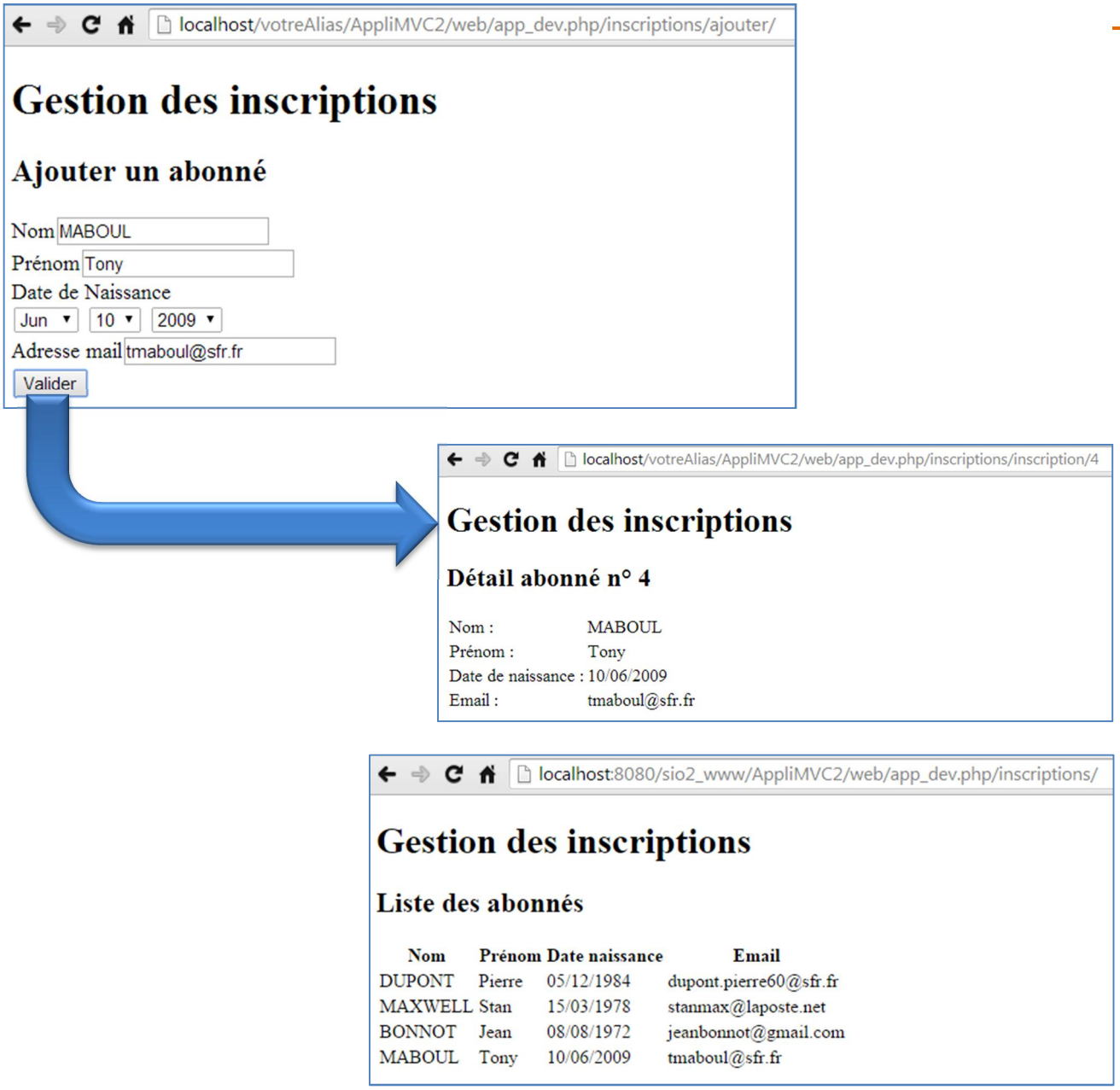
Complétez votre code comme ci-après, en lisant bien les commentaires pour comprendre ce que l'on fait. N'hésitez pas à poser des questions si cela vous semble obscur.

```
public function ajouterAction()
{
    // On crée un objet Abonne
    $abonne = new Abonne();

    // On crée un FormBuilder et on lui ajoute les champs que l'on souhaite
    $formBuilder = $this->createFormBuilder($abonne)
        ->add('nom', 'text', array('label' => 'Nom'))
        ->add('prenom', 'text', array('label' => 'Prénom'))
        ->add('dateNaissance', 'date', array('label' => 'Date de Naissance'))
        ->add('email', 'email', array('label' => 'Adresse mail'))
        ->add('Valider', 'submit');

    // On génère le formulaire à partir du formBuilder
    $form = $formBuilder->getForm();

    // On récupère la requête
    $request = $this->get('request');

    // On vérifie qu'elle est de type POST
    if ($request->getMethod() == 'POST') {

        // On fait le lien Requête <-> Formulaire
        // À partir de maintenant, la variable $abonne contient les
        // valeurs entrées dans le formulaire par le visiteur
        $form->bind($request);

        // On vérifie que les valeurs entrées sont correctes
        if ($form->isValid()) {
            // On enregistre notre objet $film dans la base de données
            $em = $this->getDoctrine()->getManager();
            $em->persist($abonne); // On « persiste » l'entité
            $em->flush(); // On « flush » tout ce qui a été persisté avant

            // On redirige vers la page de visualisation de l'abonne'
            // nouvellement créé
            return $this->redirect( $this->generateUrl('sioinscriptions_voir',
                array('id' => $abonne->getId())) );
        }
    }

    // On passe la méthode createView() du formulaire à la vue afin qu'elle
    // puisse l'afficher
    return $this->render('SioInscriptionsBundle:Inscriptions:ajouter.html.twig',
        array('form' => $form->createView()));
}
```

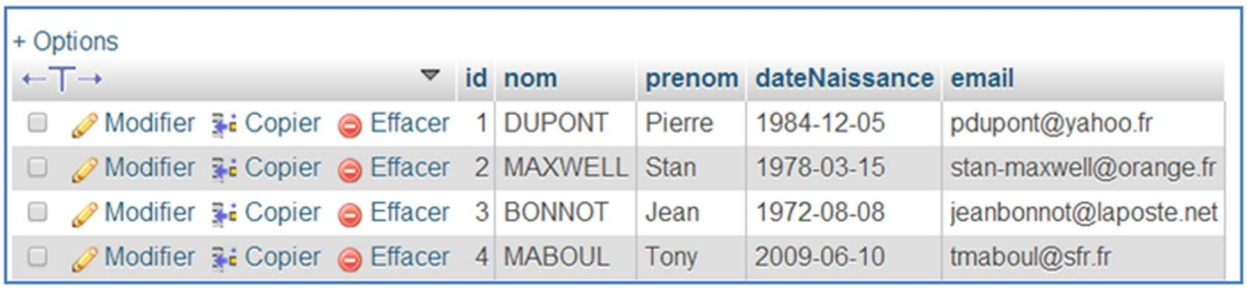
La méthode isValid() permet de vérifier que les données saisies sont correctes. Pour l'instant nous conservons de fonctionnement de base : Symfony2 effectue en effet un certain nombre de contrôles dépendants du type des champs et des options que l'on a spécifié au FormBuilder. Nous verrons un peu plus tard, comment ajouter ses propres contrôles.

Notez la manière d'effectuer une redirection vers une autre page.

Un petit test pour vérifier que tout fonctionne correctement :



On vérifie également dans la base de données :



On teste également un cas d'erreur :

Alors, qu'en dites-vous ? Ça vaut le coup de s'investir un peu, tant que le Framework MVC que sur l'ORM, n'est-ce pas ?

- Ajoutez un message flash

On va améliorer un peu notre interface, et afficher un message indiquant que l'abonné a bien été ajouté. Pour cela 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 enregistré » par exemple) puis redirige vers la page de visualisation de l'abonné nouvellement créé. Sur cette page, le message flash s'affiche puis est détruit de la session. Ainsi, si l'on change de page ou qu'on l'actualise, le message flash ne sera plus présent.

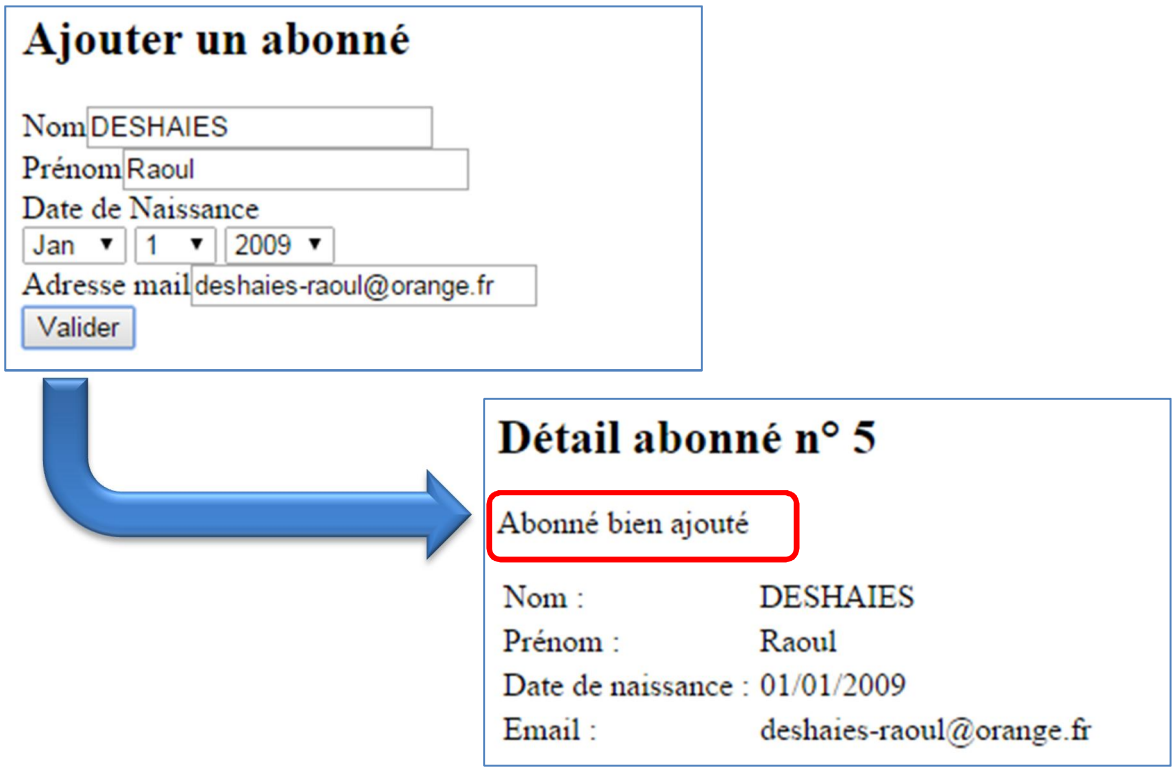
- Modifiez le code de votre méthode `ajouterAction()` comme ci-dessous :

```
public function ajouterAction()
{
    [...]
        // On vérifie que les valeurs entrées sont correctes
        if ($form->isValid()) {
            // On enregistre notre objet $film dans la base de données
            $em = $this->getDoctrine()->getManager();
            $em->persist($abonne); // On « persiste » l'entité
            $em->flush();// On « flushe » tout ce qui a été persisté avant

            // On définit un message flash
            $this->get('session')->getFlashBag()->add('info',
                'Article bien ajouté');

            // On redirige vers la page de visualisation de l'abonne'
            // nouvellement créé
            return $this->redirect($this->generateUrl('sioinscriptions_voir',
                array('id' => $abonne->getId())) );
        }
    [...]
}
```

- Modifiez le code de votre vue `voir.html.twig` comme ci-dessous :

```
{% extends "::layout.html.twig" %}

{% block stylesheets %}
{% endblock %}

{% block body %}
    <h2>Détail abonné n° {{ abonne.id }}</h2>

    <p>
        {# On affiche tous les messages flash dont le nom est "info" #}
        {% for message in app.session.flashbag.get('info') %}
            <p>{{ message }}</p>
        {% endfor %}
    </p>

<table>
[…]
```

- Testez

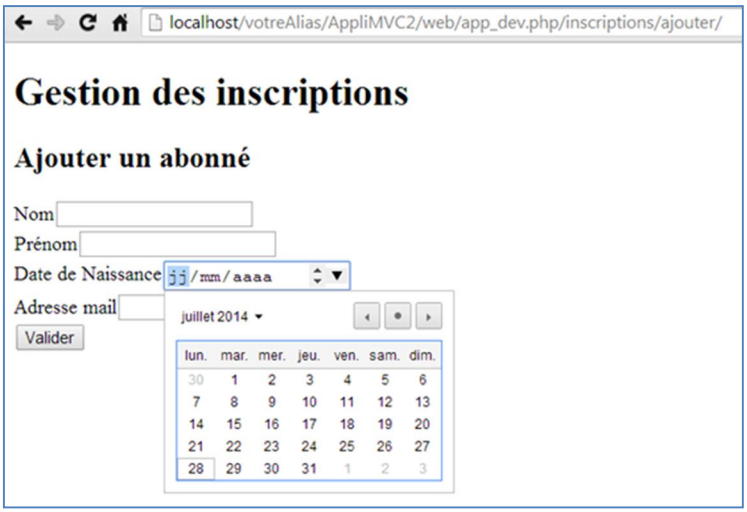


➢ Format de la date dans le formulaire

Par défaut, Symfony propose les dates la forme de 3 listes déroulantes.
De plus, on est limité à 10 années.

➢ Pour obtenir un champ de type « text », il faut ajouter une option dans le tableau des options

```
$formBuilder = $this->createFormBuilder($abonne)
    ->add('nom', 'text', array('label' => 'Nom'))
    ->add('prenom', 'text', array('label' => 'Prénom'))
    ->add('dateNaissance', 'date', array(
        'label' => 'Date de Naissance',
        'widget' => 'single_text'))
    ->add('email', 'text', array('label' => 'Adresse mail'))
    ->add('Valider', 'submit');
```



➢ Pour conserver les listes déroulantes, mais en ayant plus d'années dispo

Il faut utiliser le type de widget « birthday » au lieu de « date »

```
$formBuilder = $this->createFormBuilder($abonne)
    ->add('nom', 'text', array('label' => 'Nom'))
    ->add('prenom', 'text', array('label' => 'Prénom'))
    ->add('dateNaissance', 'birthday', array(
        'label' => 'Date de Naissance'))
    ->add('email', 'text', array('label' => 'Adresse mail'))
    ->add('Valider', 'submit');
```

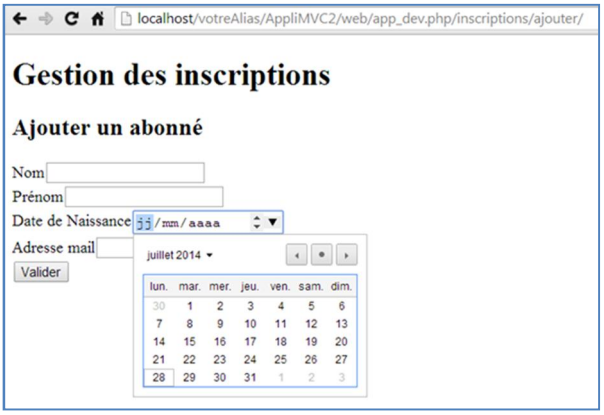
➢ Pour obtenir les noms des mois en français

Il faut modifier la valeur du paramètres locale dans le fichier /app/config/parameters.yml

```
# This file is auto-generated during the composer install
parameters:
    database_driver: pdo_mysql
    database_host: 127.0.0.1
    database_port: null
    database_name: tpmvc2
    database_user: root
    database_password: null
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    mailer_user: null
    mailer_password: null
    locale: fr
    secret: ThisTokenIsNotSoSecretChangeIt
    debug_toolbar: true
    debug_redirects: false
    use_assetic_controller: true
```

Vous pouvez remarquer que par la même occasion, les 3 listes déroulantes ont changé de position.

Apporter les modifications nécessaires pour que la date soit gérée sous la forme d'un champ de type text et que les noms des mois soient en français.



Action supprimer

Modification du contrôleur

Pour cette action, on va utiliser un formulaire vide.

Enfin, pas totalement vide, puisque tous les formulaires Symfony contiennent un champ CSRF, permettant d'éviter les failles CSRF (faire supprimer un abonné à un utilisateur sans que celui ne le sache).

Pour plus d'informations que les failles CSRF (Cross-Site Request Forgery) :
http://fr.wikipedia.org/wiki/Cross-site_request_forgery

Voilà donc le code du l'action : rien de bien nouveau, si ce n'est que l'on génère cette fois-ci un formulaire vide.

```
public function supprimerAction($id)
{
    // On récupère l'abonné
    $abonne = $this->getDoctrine()
                ->getManager()
                ->getRepository('SioInscriptionsBundle:Abonne')
                ->find($id);

    // Si aucun abonné n'a été trouvé avec l'id $id, on lève une exception
    if ($abonne == null)
    {
        throw $this->createNotFoundException("Abonné[id='$id] inexistant.");
    }

    // On crée un formulaire vide, qui ne contiendra que le champ CSRF
    // Cela permet de protéger la suppression d'abonnés contre cette faille
    $form = $this->createFormBuilder()->getForm();

    $formBuilder = $this->createFormBuilder($abonne)
        ->add('Supprimer', 'submit');

    // On génère le formulaire à partir du formBuilder
    $form = $formBuilder->getForm();

    $request = $this->getRequest();
    if ($request->getMethod() == 'POST') {
        $form->bind($request);
        if ($form->isValid()) {
            // On supprime l'abonné
            $em = $this->getDoctrine()->getManager();
            $em->remove($abonne);
            $em->flush();

            // On définit un message flash
            $this->get('session')->getFlashBag()->add('info', 'Abonné bien supprimé');

            // Puis on redirige vers l'accueil
            return $this->redirect($this->generateUrl('sioinscriptions_accueil'));
        }
    }

    // La requête est en GET, on affiche une page de confirmation avant de supprimer
    return $this->render('SioInscriptionsBundle:Inscriptions:supprimer.html.twig',array(
        'abonne' => $abonne,
        'form' => $form->createView()
    ));
}
```

➢ Modification de la vue (supprimer.html.twig)

```
{% extends "::layout.html.twig" %}

{% block stylesheet %}
{% endblock %}

{% block body %}
    <h2>Supprimer un abonné</h2>

    <p>Etes-vous certain de vouloir supprimer l'abonné "{{ abonne.nom }}
       {{ abonne.prenom }}" ?</p>

    {{ form_start(form) }}
        {{ form_widget(form) }}
    {{ form_end(form) }}
{% endblock %}

{% block javascripts %}
{% endblock %}
```

➢ Affichage du message flash sur la page accueil (vue index.html.twig)

```
{% extends "::layout.html.twig" %}

{% block stylesheet %}
{% endblock %}

{% block body %}
    <h2>Liste des abonnés</h2>

    <p>
        {# On affiche tous les messages flash dont le nom est « info » #}
        {% for message in app.session.flashbag.get('info') %}
            <p>{{ message }}</p>
        {% endfor %}
    </p>

    <table class="table table-striped">
    [...]
```

Vous devez obtenir :

A vous ...

On veut pouvoir modifier un abonné.

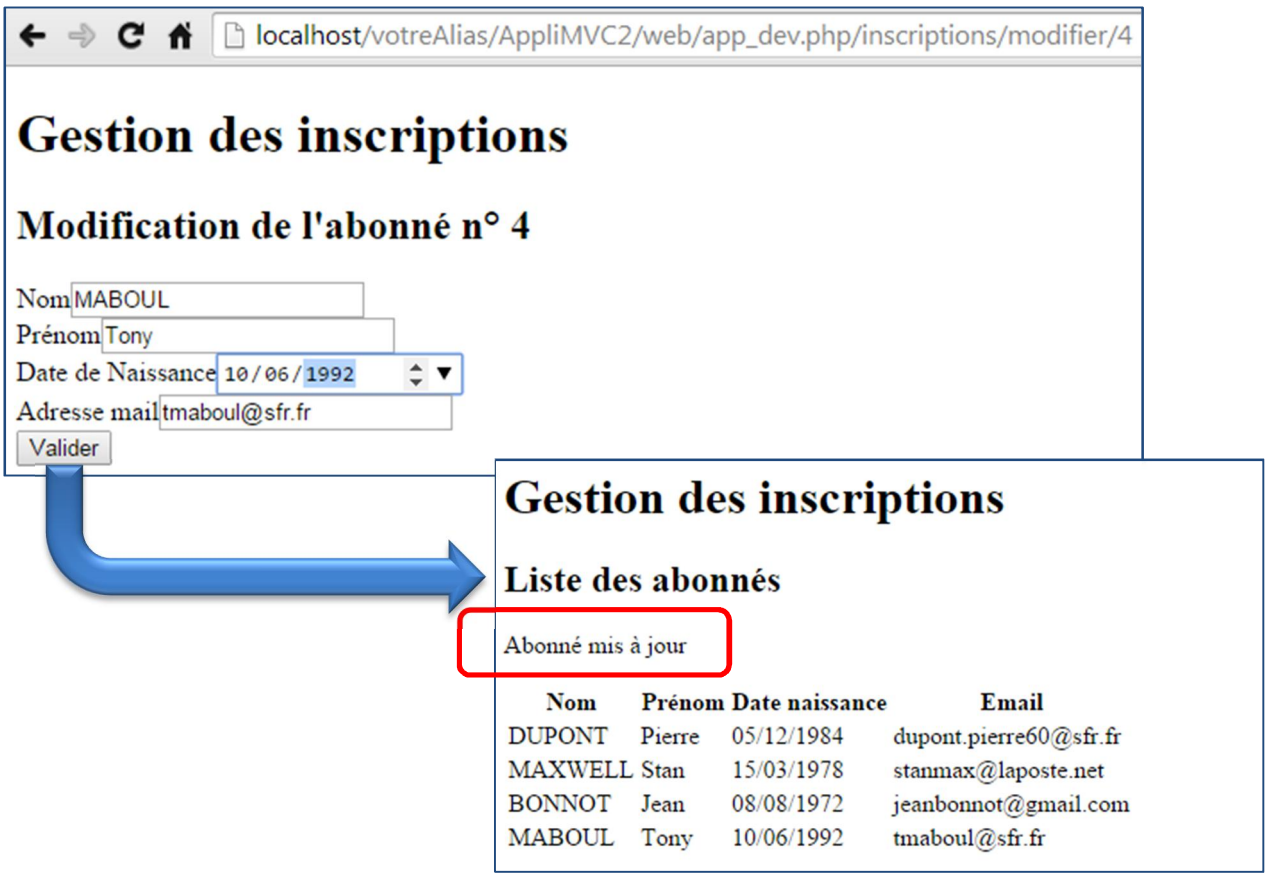
Ajouter une route à votre bundle

Créer la méthode `modifierAction()` dans la classe de votre contrôleur

Le code est identique à celui de la méthode `ajouterAction()` hormis quelques petites adaptations :

- Remplacer : `$abonne = new Abonne();`
  par les instructions permettant de récupérer les informations de l'abonné dont l'id est passé en paramètre (aidez-vous de la méthode `voirAction()`).
- Il faut passer l'objet Abonne à la méthode `createFormBuilder()`
- Changer le nom de la vue dans l'appel de la méthode render. Attention, en plus de passer le formulaire il faut également passer l'abonné !
- On redirige ensuite vers la page contenant la liste de tous les abonnés.

Vous devez obtenir :



Testez.