Symfony2 utilise **le composant Form**, réutilisable dans d'autres projets même sans symfony.
Un formulaire Symfony se construit sur un objet existant, et son but est d'hydrater cet objet, c-à-d que c'est le formulaire qui va remplir les attributs de l'objet avec les valeurs entrées par l'utilisateur.

1. CREATION FORMULAIRE AJOUT D'UN STAGE

1.1. Création de la route

```
Stage_ajouter:
    path:      /ajouterStage
    defaults:  { _controller: webStageEtudiantBundle:Etudiant:ajouterStage }
```

1.2. Ajout de la méthode correspondante dans le controleur

Pour créer un formulaire il faut 2 éléments :
- un objet
- un constructeur de formulaire → le FormBuilder

Pour ajouter des champs à un formulaire, il faut utiliser la méthode <?php $formBuilder->add() du FormBuilder. Les arguments sont les suivants :
1. Le nom du champ ;
2. Le type du champ (date, text, number, percent, ..., voir annexe1). Il est primordial de bien faire correspondre les types de champ du formulaire avec les types d'attributs que contient votre objet.
3. Les options du champ, sous forme de tableau. Les options permettent par exemple de rendre le champ facultatif, de modifier l'apparence de la date.

On utilisera cette méthode également pour imbriquer des formulaires :

```
 ->add('autreFormulaire', new autreFormulaireType());
```

```
public function ajouterStageAction()
  {
    $stage = new Stage();
       // On crée le FormBuilder grâce à la méthode du contrôleur createFormBuilder
       // equivaut à dire de créer un formulaire autour de l'objet $stage
       $formBuilder = $this->createFormBuilder($stage);

  // On ajoute les champs de l'entité que l'on veut à notre formulaire
  $formBuilder
             ->add('libelle',       'text', array('required' => false))
             ->add('dateDebut',        'date',array(
                                                 'input'  => 'datetime',
                                                 'widget' => 'single_text',
                                                 'format' => 'dd/MM/yyyy'))
             ->add('dateFin',        'date')
             ->add('note',       'text');

       // À partir du formBuilder, on génère le formulaire
       $form = $formBuilder->getForm();

       // On passe la méthode createView() du formulaire à la vue afin qu'elle puisse afficher le
       formulaire toute seule
       return $this->render('webStageEtudiantBundle:Etudiant:ajouterStage.html.twig', array(
       'form' => $form->createView(),
                                      ));

       // à compléter pour soumettre le formulaire voir ci-dessous.

  }
```

1.3. création de la vue formulaire nommée ajouterStage.html.twig

La fonction Twig {{ form_widget() }}permet d'afficher un formulaire entier en une seule ligne.
Il faut cependant ajouter la balise <form> et le bouton de soumission car cette fonction n'affiche que l'intérieur du formulaire.

```
{# src/webStage/EtudiantBundle/Resources/views/Etudiant/ajouterStage.html.twig #}

<h3>Formulaire de stage</h3>
<div class="well">
  <form method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}
    <input type="submit" class="btn btn-primary" />
  </form>
</div>
```

Note : si vous avez utiliser un constructeur pour votre entité, votre formulaire sera déjà pré-rempli avec les données passées dans le constructeur.

2. GENERATION DU FORMULAIRE AVEC LA CONSOLE

Il est possible de déplacer la construction du formulaire, du contrôleur à une classe externe. Le formulaire généré correspond alors à la définition des champs de notre formulaire. Ainsi, si l'on utilise le même formulaire sur plusieurs pages différentes, on utilisera ce même formulaire type. Le formulaire est donc réutilisable.
Cela permet de plus, d'épurer le controleur des instructions de construction du formulaire.

2.1. Creation du formulaireType Etudiant

On peut utiliser la console pour créer un formulaire réutilisable avec la commande Doctrine suivante :

```
C:\wamp\www\Symfony>php app/console doctrine:generate:form webStageEtudiantBundle:Etudiant
The new EtudiantType.php class file has been created under C:\wamp\www\Symfony\src\webStage\EtudiantBundl
e/Form/EtudiantType.php.
```

La classe EtudiantType est générée dans C:\wamp\www\Symfony\src\webStudent\EtudiantBundle\Form
Dans cette nouvelle classe, les types de champs ne sont pas précisés. Le composant Form va les deviner à partir des annotations Doctrine qu'on a mis dans l'objet.
Il est cependant vivement recommandé de **préciser les types de champs**.

Si certains champs générés automatiquement ne nous intéresse pas, il est possible de les supprimer

```
$builder->remove('section');
```

2.2. methode ajouterEtudiantAction (la route doit exister)

Pour construire le formulaire créer précédemment, il suffit alors juste d'ajouter les 2 premières lignes dans la fonction du controleur ci-dessous.

```
public function ajouterEtudiantAction()
{
      $etudiant = new Etudiant ();
      $form = $this->createForm(new EtudiantType, $etudiant);

      /**
      ***    INSTRUCTIONS DE SOUMISSION DU FORMULAIRE
      **/

      // 1. On récupère la requête HTTP
      $request = $this->get('request');

    // 2. On vérifie qu'elle est de type POST
    if ($request->getMethod() == 'POST') {
      // 3. On fait le lien Requête <-> Formulaire
```

```
      // À partir de maintenant, la variable $etudiant 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 $etudiant dans la base de données
        $em = $this->getDoctrine()->getManager();
        $em->persist($etudiant);
        $em->flush();

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

    // À ce point là :
    // - Soit la requête est de type GET, donc le visiteur vient d'arriver sur la page et veut voir
le formulaire
    // - Soit la requête est de type POST, mais le formulaire n'est pas valide, donc on l'affiche
de nouveau

    return $this->render('webStageEtudiantBundle:Etudiant:ajouterEtudiant.html.twig', array(
    'form' => $form->createView(),

     ));
  }
```

2.3. Gestion de la soumission d'un formulaire

Code ci-dessus en gras

- Pour gérer l'envoi du formulaire, il faut tout d'abord vérifier que la requête est de type POST : cela signifie que le visiteur est arrivé sur la page en cliquant sur le bouton `submit` du formulaire. Lorsque c'est le cas, on peut traiter notre formulaire.

- Ensuite, il faut faire le lien entre les variables de type POST et notre formulaire, pour que les variables de type POST viennent remplir les champs correspondants du formulaire. Cela se fait via la méthode `bind()` du formulaire. Cette méthode dit au formulaire : « Voici la requête d'entrée (nos variables de type POST entre autres). Lis cette requête, récupère les valeurs qui t'intéressent et hydrate l'objet. » Comme vous pouvez le voir, elle fait beaucoup de choses !

- Enfin, une fois que notre formulaire a lu ses valeurs et hydraté l'objet, il faut tester ces valeurs pour vérifier qu'elles sont valides avec ce que l'objet attend. Il faut *valider* notre objet. Cela se fait via la méthode `isValid()` du formulaire.

2.4. Creation du formulaire etudiant.html.twig

```
{# src/webStage/EtudiantBundle/Resources/views/Etudiant/ajouterEtudiant.html.twig #}

<h3>Formulaire de etudiant</h3>

<div class="well">
  <form method="post" {{ form_enctype(form) }}>
```

```
      {{ form_widget(form) }}

    <input type="submit" class="btn btn-primary" />
  </form>
</div>
```

3. Personnalisation des formulaires

Si le formulaire généré ne convient pas, il est toujours de le personnaliser dans la vue twig.

```
../
<div>
{# Exemple pour un champ #}

{# Génération du label. #}
{{ form_label(form.nom, "Nom      ") }}

{# Affichage des erreurs pour ce champ précis. #}
{{ form_errors(form.nom) }}

{# Génération de l'input. #}
{{ form_widget(form.nom, {'attr': {'placeholder': 'nom'} }) }}
</div>
/..
```

4. Valider les données d'un formulaire

Pour définir les règles de validation, nous allons donc utiliser les annotations (il est possible aussi d'externaliser les règles de validation dans un fichier extérieur).
Pour cela , nous utiliserons donc @Assert, donc le namespace complet est le suivant :

```
<?php
use Symfony\Component\Validator\Constraints as Assert;
```

Les annotations concernant les règles de validation sont à ajouter dans les entités, au niveau de chaque attribut avec la syntaxe suivante :

```
@Assert\Contrainte(valeur de l'option par défaut)
```

Exemples

```
/**
* @Assert\NotBlank()
* @Assert\MinLength(3)
* @Assert\DateTime()
* @Assert\MinLength(limit=10, message="Le nom  doit faire au moins {{ limit }} caractères ;")
* @Assert\Max(2000)
```

Lorsque l'on manipule les formulaires,la méthode($form->isValid()) permet de valider automatiquement les données avec les asserts définis.
Il est aussi possible de valider directement un objet avec le service validator.

Il est également possible de créer ses propres règles de validation.
Voici la documentation officielle sur les contraintes.
http://symfony.com/fr/doc/current/reference/constraints.html

5. ANNEXE 1

Texte	Choix	Date et temps	Divers	Multiple	Caché
text					
textarea					
email	choice				
integer	entity	date			
money	country	datetime	checkbox	collection	hidden
number	language	time	file	repeated	csrf
password	locale	birthday	radio		
percent	timezone				
search					
url					