1. LE ROUTEUR SYMFONY

1.1. RÔLE DU ROUTEUR

Le rôle du routeur est, à partir d'une URL, de déterminer quel contrôleur appeler et avec quels arguments.
L'objectif est de créer ce que l'on appelle un fichier de mapping des routes (un fichier de correspondances, en français). Ce fichier, généralement situé dans votreBundle/Resources/config/routing.yml, contient la définition des routes. Chaque route fait la correspondance entre une URL et le contrôleur à appeler.

Ajout des routes dans C:\wamp\www\Symfony\src\webStudent\EtudiantBundle\Resources\config\routing.yml

```
Etudiant_accueil:
    path:      /etudiant
    defaults:  { _controller: webStudentEtudiantBundle:Etudiant:index }
Etudiant_consulter:
    path:      /etudiant/stage/{id}
    defaults:  { _controller:webStudentEtudiantBundle:Etudiant:consulter }
Etudiant_ajouter:
    path:      /etudiant/ajouter
    defaults:  { _controller: webStudentEtudiantBundle:Etudiant:ajouter }
Etudiant_modifier:
    path:      /etudiant/modifier
    defaults:  { _controller: webStudentEtudiantBundle:Etudiant:modifier }
```

1.2. Fonctionnement du routeur

Dans le code précédent, vous pouvez distinguer 4 blocs. Chacun correspond à une route. Nous les verrons en détail plus loin, mais vous pouvez constater que chaque route prend :

- Une entrée (ligne `path`) : c'est l'URL à capturer ;
- Une sortie (ligne `defaults`) : ce sont les paramètres de la route, notamment celui qui dit quel est le contrôleur à appeler.

Le but du routeur est donc, à partir d'une URL, de trouver la route correspondante et de retourner le contrôleur que veut cette route. Pour trouver la bonne route, le routeur va les parcourir une par une, dans l'ordre du fichier, et s'arrêter à la première route qui fonctionne.

1.3. Composition d'une route :

Elle est constituée au minimum de trois éléments :

- `Etudiant_accueil` est le nom de la route. Il n'a aucune importance dans le travail du routeur, mais il interviendra lorsque l'on voudra générer des URL. Il faut qu'un nom soit unique et clair. On a donc préfixé les routes de « `Etudiant` » pour plus de clarté entre bundles.
- `path: /etudiant` est l'URL sur laquelle la route s'applique. Ici, « / `etudiant` » correspond à une URL absolue du type `http://www.monsite.com/etudiant`.
- `defaults: { _controller: webStudentEtudiantBundle:Etudiant:index }` correspond au contrôleur à appeler.

1.4. Créer une route avec des paramètres

```
Etudiant_consulter:
    path:      etudiant/consulter/{id}
    defaults:  { _controller: webStudentEtudiantBundle:Etudiant:etudiantConsulter }
```

id est un paramètre, donc modifiable

Ajouter dans le controlleur, la méthode consulterAction

```
public function consulterAction($id)
 {
   // $id vaut 5 si l'on a appelé l'URL /Etudiant/stage/5
   // Ici, on récupèrera depuis la base de données l'étudiant correspondant à l'id $id
   // Puis on passera l'etudiant à la vue pour qu'elle puisse l'afficher
     return new Response("Affichage de l'étudiant d'id : ".$id.".");
 }
```

Tests :
http://localhost/symfony/web/app_dev.php/etudiant/consulter/5

1.5. ACTIVITE :

1. Dans votre bundle correspondant à la gestion des stages, ajouter la route et la méthode dans le controleur qui permettra d'afficher un stage. Le numéro de stage sera passé en paramètre.
http://localhost/symfony/web/app_dev.php/stage/consulter/5

2. Dans le bundle etudiant, ajouter une route et une methode affichant deux paramètres : le premier étant la promo, le second l'id etudiant.
http://localhost/symfony/web/app_dev.php/etudiant/consulter/2013/7

1.6. Créer une route avec des contraintes

Dans notre exemple de route et d'url, rien n'empêche de saisir
http://localhost/symfony/web/app_dev.php/etudiantPromo/anneeFolle/etudiantZero
au lieu
http://localhost/symfony/web/app_dev.php/etudiantPromo/2013/7
ce qui ramènerait une erreur 404.

Pour éviter ceci, nous pouvons ajouter des contraintes à l'aide du mot clé requirement et des expressions régulières :
Exemple :

```
Etudiant_consulterPromo:
    path:      /etudiantPromo/{promo}/{id}
    defaults:  { _controller:webStudentEtudiantBundle:Etudiant:consulterPromo }
    requirements:
        promo: \d{4}
        id:  \d{1}
```

1.7. Générer des url

1.7.1. Générer une url depuis le contrôleur

Pour générer une URL, vous devez le demander au routeur en lui donnant deux arguments : le nom de la route ainsi que les éventuels paramètres de cette route.
Exemple dans le controleur Etudiant

```
return $this->redirect( $this->generateUrl('Etudiant_consulter', array('etudiant' => $etudiantId->getId(),

                                'etudiantNom'=> $etudiant->getNom() )) );
```

1.8. Générer une url depuis une vue twig

Vous aurez bien plus l'occasion de devoir générer une URL depuis une vue. C'est la fonction path qu'il faut utiliser depuis un template Twig :

```
<a href="{{ path('Etudiant_consulter', { 'id': etudiant_id }) }}">Lien vers etudiant d'id {{ etudiant_id }}</a>
```

1.9. En résumé

- Une route est composée au minimum de deux éléments : l'URL à faire correspondre (son `path`), et le contrôleur à exécuter (attribut `_controller`).
- Le routeur essaie de faire correspondre chaque route à l'URL appelée par l'internaute, et ce dans l'ordre d'apparition des routes : la première route qui correspond est sélectionnée.
- Une route peut contenir des arguments, facultatifs ou non, représentés par les accolades {argument}, et dont la valeur peut être soumise à des contraintes via la section `requirements`.
- Le routeur est également capable de générer des URL à partir du nom d'une route, et de ses paramètres éventuels.

2. LES CONTROLEURS

Le contrôleur contient toute la logique de votre application. En fait, il ne fait qu'utiliser des services, les modèles et retourner une **réponse**. Cette réponse peut être de retourner une vue. Finalement, c'est un chef d'orchestre qui se contente de faire la liaison entre tout le monde.

Il existe dans Symfony2 une classe Response. Retourner une réponse signifie donc tout simplement : instancier un objet Response, que l'on peut appeler $response, et faire un return $response.

Exemple :

```
class EtudiantController extends Controller
{
  public function indexAction()
  {
    //return new Response("Salut tout le monde, lebes ?");
      return $this->render('webStudentEtudiantBundle:Etudiant:index.html.twig');
  }
```

2.1. Les paramètres contenus dans les routes

La première manière de récupérer des arguments est de récupérer ceux contenus dans la route (voire précédemment).

Exemple - Fichier controleur

```
public function consulterAction($id)
  {
      return new Response("Affichage de l'étudiant d'id : ".$id.".");
  }
```

La route associée

```
Etudiant_consulter:
    path:      /stage/{id}
    defaults:  { _controller:webStudentEtudiantBundle:Etudiant:consulter }
```

2.2. Les paramètres hors routes

Dans le fichier controleur

```
public function consulterAction($id)
  {
    // On récupère la requête
    $request = $this->getRequest();
    // On récupère notre paramètre monIdEtudiant
    $monIdEtudiant = $request->query->get('monIdEtudiant');
    return new Response("Affichage de l'etudiant d'id : ".$id.", avec le parametre passe
dans la request : ".$monIdEtudiant);

  }
```

Route associée

```
Etudiant_consulter:
    path:       /stage/{id}
    defaults:  { _controller:webStudentEtudiantBundle:Etudiant:consulter }
```

Attention : Avec cette façon d'accéder aux paramètres, vous n'avez pas besoin de tester leur existence. Par exemple, si vous faites $request->query->get('paramVide') alors que le paramètre paramVide n'est pas défini dans l'URL, cela vous retournera une chaîne vide, et non une erreur

2.3. Réponse et Vues

Généralement, vous préférerez que votre réponse soit contenue dans une vue, dans un template. Heureusement pour nous, le contrôleur dispose d'un raccourci : la méthode <?php $this->**render()**.
Elle prend en paramètres le nom du template et ses variables, puis s'occupe de tout : créer la réponse, y passer le contenu du template, et retourner la réponse.
Exemple - dans le contrôleur

```
return $this->render('webStudentEtudiantBundle:Etudiant:index.html.twig', array('id'  => $id
 ));
```

Dans la vue avec recup du parametre

```
<body>
    <h1>Salut les étudiants ayant pour id {{ id }}</h1>
```

2.4. Réponse et redirection

Vous serez sûrement amenés à faire une redirection vers une autre page. Or notre contrôleur est obligé de retourner une réponse. Dans ce cas, il faut faire une redirection.
$this->redirect(). Elle prend en paramètre l'URL vers laquelle vous souhaitez faire la redirection et s'occupe de créer une réponse.

2.5. Utiliser des services : la session

Les outils de session sont également intégrés dans un service (un objet qui nous rend services à travers ses méthodes).

```
// Récupération du service
    $session = $this->get('session');

    // On récupère le contenu de la variable user_id
    $user_id = $session->get('user_id');

    // On définit une nouvelle valeur pour cette variable user_id
    $session->set('user_id', 91);
```

Un autre outil très pratique du service de session est ce que l'on appelle les « messages flash ». Un terme précis pour désigner en réalité une variable de session qui ne dure que le temps d'une seule page. C'est une astuce utilisée pour les formulaires par exemple : la page qui traite le formulaire définit un message flash (« Etudiant bien enregistré » par exemple) puis redirige vers la page de visualisation de l'étudiant nouvellement créé. Sur cette page, le message flash s'affiche, et est détruit de la session. Alors si l'on change de page ou qu'on l'actualise, le message flash ne sera plus présent. Voir un exemple d'utilisation sur le site du zéro.

3. LE MOTEUR DE TEMPLATE TWIG

Les templates vont nous permettre de séparer le code PHP du code HTML/XML/Text, etc. Seulement, pour faire du HTML de présentation, on a toujours besoin d'un peu de code dynamique : faire une boucle pour afficher tous les articles d'un blog, créer des conditions pour afficher un menu différent pour les utilisateurs authentifiés ou non, etc. Pour faciliter ce code dynamique dans les templates, le moteur de templates Twig offre son pseudo-langage à lui. Ce n'est pas du PHP, mais c'est plus adapté et voici pourquoi :

- La syntaxe est plus concise et plus claire. Pour afficher une variable, `{{ mavar }}` suffit, alors qu'en PHP il faudrait faire `<?php echo $mavar; ?>`.
- Il y a quelques fonctionnalités en plus, comme l'héritage de templates (voir ci-dessous).
- Il sécurise vos variables automatiquement : plus besoin de se soucier de `htmlentities()`, `addslashes()`, `etc...`

3.1. Syntaxe de base

Src :http://fr.openclassrooms.com/informatique/cours/developpez-votre-site-web-avec-le-framework-symfony2/afficher-des-variables)

Description	Exemple Twig	Équivalent PHP
Afficher une variable	Pseudo : {{ pseudo }}	Pseudo : <?php echo $pseudo; ?>
Afficher l'index d'un tableau	Identifiant : {{ user['id'] }}	Identifiant : <?php echo $user['id']; ?>
Afficher l'attribut d'un objet, dont le getter respecte la convention $objet->getAttribut()	Identifiant : {{ user.id }}	Identifiant : <?php echo $user->getId(); ?>
Afficher une variable en lui appliquant un filtre. Ici, « upper » met tout en majuscules :	Pseudo en majuscules : {{ pseudo|upper }}	Pseudo en lettre majuscules : <?php echo strtoupper($pseudo); ?>
Afficher une variable en combinant les filtres. « striptags » supprime les balises HTML. « title » met la première lettre de chaque mot en majuscule. Notez l'ordre d'application des filtres, ici striptags est appliqué, puis title.	Message : {{ news.texte|striptags|title }}	Message : <?php echo ucwords(strip_tags($news->getTexte())); ?>
Utiliser un filtre avec des arguments. Attention, il faut que date soit un objet de type Datetime ici.	Date : {{ date|date('d/m/Y') }}	Date : <?php echo $date->format('d/m/Y'); ?>
Concaténer	Identité : {{ nom ~ " " ~ prenom }}	Identité : <?php echo $nom.' '.$prenom; ?>

La documentation de tous les filtres disponibles est dans la documentation officielle de Twig : http://twig.sensiolabs.org/doc/filters/index.html.

3.2. Structures algorithmiques

- {{ … }}*affiche* quelque chose ;
- {% … %}*fait* quelque chose ;

- {# … #} n'affiche rien et ne fait rien : c'est la syntaxe pour les commentaires, qui peuvent être sur plusieurs lignes.

- Condition : {% if %}
- Boucle : {% for %}
- Définition : {% set %}

Exemples - extrait d'une vue twig

```
<h3>LISTE DES ETUDIANTS RECHERCHES</h3>
<table class="table table-bordered table-striped table-condensed">
{% for a in listeEtudiant %}
    <tr>
      <td><a href="{{ path('Etudiant_consulter', { 'id': a.id }) }}">{{ a.nom }}</a></td>
      <td><a href="{{ path('Etudiant_consulter', { 'id': a.id }) }}">{{ a.prenom }}</a></td>
      <td>{{ a.dateNaissance|date('d/m/Y') }}</td>
      <td>{{ a.adrMail}}</td>
      <td>{{ a.section.code}} - {{ a.niveau.id}} </td>
      <td>{{ a.promo.annee}}</td>
      <td> <a href="{{ path('Etudiant_modifier', { 'id': a.id }) }}"> <img src="
{{ asset('img/crayon.png') }} "> </a></td>
{% else %}
    <tr>
      <td>Aucun etudiant n'a été trouvé.</td>
    </tr>
{% endfor %}
</table>
```

3.3. Héritage de template

3.3.1. Le principe

Le principe est simple : vous avez un template père qui contient le design de votre site ainsi que quelques trous (appelés « *blocks* » en anglais, que nous nommerons « blocs » en français) et des templates fils qui vont remplir ces blocs. Les fils vont donc venir hériter du père en remplaçant certains éléments par leur propre contenu.

Exemple :

```
{# src/webStudent/EtudiantBundle/Resources/views/layout.html.twig #}

<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block title %}WEBSTUDENT{% endblock %}</title>
  </head>
  <body>

    {% block body %}
    {% endblock %}

  </body>
</html>
```

Le template fils

```
{# src/webStudent/EtudiantBundle/Resources/views/Etudiant/listeEtudiant.html.twig #}
```

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

{% block title %} {{ parent() }} - Index {% endblock %}

{% block body %}
    C'est ici qu'on ré-écrit tout le contenu, les autres balises étant héritées du layout !
{% endblock %}
```

3.3.2. Le modèle « triple héritage »

Pour bien organiser ses templates, une bonne pratique est sortie du lot. Il s'agit de faire de l'héritage de templates sur trois niveaux, chacun des niveaux remplissant un rôle particulier. Les trois templates sont les suivants :

- Layout général : c'est le design de votre site, indépendamment de vos bundles. Il contient le header, le footer, etc. La structure de votre site donc (c'est notre template père).
  Le layout général ne dépendant pas d'un bundle en particulier, il faut le mettre dans le répertoire `app/Resources/views/layout.html.twig`.
  Et pour l'appeler depuis vos templates, la syntaxe est la suivante : « `::layout.html.twig` ».

- Layout du bundle : il hérite du layout général et contient les parties communes à toutes les pages d'un même bundle. Par exemple, pour notre application, on pourrait afficher un menu particulier.

- Template de page : il hérite du layout du bundle et contient le contenu central de votre page.