Rapport de stage

Projet L@mbda

Session 2016 / 2017

Greta Vannes

Sommaire

Présentation du projet

But du projet

Le but de ce projet est de créer une application capable de gérer des collections d'objets, avec des fonctionnalités bien spécifiques. En effet, il s'agit de permettre à une communauté d'utilisateurs de créer des collections d'objets, et de mettre à disposition ces objets au travers d'emprunts et de prêts, à tous les autres utilisateurs de la communauté. Chacun pourra gérer sa propre collection d'objets, et chacun pourra avoir des exemplaires différents des mêmes objets. Les utilisateurs pourront se regrouper en groupe ou associations, et pourront gérer leur propre groupe.

Fonctionnalités souhaitées

- Gestion des utilisateurs
- Gestion des adresses utilisateurs
- Gestions des objets
- Gestions des exemplaires
- Gestion de l'état des exemplaires
- Gestions des emprunts d'exemplaires appartenant à un utilisateur
- Gestions des catégories d'objets
- Gestion des sous-catégories d'objets
- Gestion des messages (au sens large)
- Gestion des commentaires (sur objet, sur utilisateur, sur emprunt)
- Gestion des notes (sur objet, sur utilisateur, sur emprunt)
- Gestion des contacts
- Gestion des groupes
- Gestion des rendez-vous
- Gestion des photos
- Gestion des éventuels événements

Tout un programme !

Réflexions sur les fonctionnalités

Au vu du but de l'application, et des fonctionnalités à y implémenter, il est nécessaire de réfléchir à une organisation, et au contenu des fonctionnalités. Autrement dit, en clair, qu'est-ce qu'on veut....

Voyons cela du point de vue de l'utilisateur, en gardant une petite touche de logique de programmation.

On veut une application, et la base de données associée, qui renfermera une collection d'objets divers et variés, regroupés en catégories, qui elles-même donneront accès à des sous catégories, et à des propriétés spécifiques à cette catégorie d'objets (exemple : un objet à un nom et une description; un livre aura, en plus, un numéro ISBN, un auteur, etc.). Ces objets seront considérés comme des objets "génériques".

Les utilisateurs auront tout le loisir de pouvoir choisir un de ces objets génériques, et de signifier qu'ils en possèdent un dans leur collection. Ce faisant, ils le mettent "à disposition" de toute la communauté.

Les autres utilisateurs peuvent voir l'état de cet exemplaire, demander au propriétaire de lui emprunter, et même convenir d'un rendez-vous.

Dans le même temps, ils peuvent s'envoyer des messages, se commenter, se noter entre eux. Les utilisateurs peuvent aussi noter et commenter les objets.

Ils peuvent également former des groupes d'utilisateurs.

Vous l'avez compris, c'est ambitieux.

Technologies utilisées

Les technologies utilisées sont diverses et variées, je liste donc les principales ci-dessous:

Symfony

Doctrine2

NetBeans IDE

La plateforme

MySQL

Symfony est un framework PHP. Il implémente tout un tas d'outils afin de faciliter la vie des programmeurs, pour ne pas avoir besoin de 'réinventer la roue'.

Il a été fondé par l'agence française SensioLabs afin de permettre de mettre en commun le code qui est utilisé dans la majorité des applications (gestion des utilisateurs, persistance des objets dans les bases de données, etc.).

Ainsi, Symfony intègre un ORM, Doctrine, qui permet d'automatiser certaines tâches, notamment le mapping de la base de données avec les classes du programme.

On devine facilement ses avantages, ainsi que le gain de temps qu'il apporte, même si je vais le relativiser tout de suite.

En effet l'environnement Symfony demande un apprentissage assez conséquent, et sa prise en main peut être assez déroutante dans les premiers temps, et lors de la montée en compétence, ce qui peut générer des difficultés.

Symfony intègre par défaut 3 grandes technologies sans quoi il ne serait rien:

- Doctrine pour le côté Modèle (entités sous Symfony), et pour gérr les accès en Base de données,
- Twig, le célèbre moteur de templates pour les Vues,
- Et un routeur, qui gèrera le routing de l'application, intégrant un firewall afin de gérer les droits d'accès aux différentes parties de l'application.

À noter aussi que Symfony est parfaitement couplé avec Composer, qui permet d'installer les composants qui pourraient être nécessaires.

Les Modèles

Doctrine

C'est sans doute la plus grande force de Symfony : Doctrine.

Cette couche logicielle gère tout ce qui a trait à la base de données, à la lecture des données qui s'y trouvent, ainsi qu'à la persistance en base des objets PHP qu'on manipule.

Il y a deux approches. Elles se valent toutes les deux. Une prend moins de temps mais est plus difficile, l'autre prend plus de temps mais on est sûr de ce qu'on fait.

- En effet, soit vous créez une base de données, les tables, clés, etc., et laissez Doctrine générer et gérer les entités (classes modèle) à partir de cette base, et les relations entre les entités;
- Soit vous créez vous-même les classes modèles, et Doctrine, en lui indiquant une base de données vierge, créera toutes les tables et clés pour vous !

Comment ça marche

Et bien c'est très simple une fois quelques règles bien comprises. Doctrine utilise plusieurs formats d'écriture pour la définition des entités (les classes et les objets dérivés de ces classes). Il y en a beaucoup, les plus utilisés étant le format de fichier yaml, le xml bien entendu, et enfin les 'annotations'.

Les annotations sont recommandées, parce que la définition des entités est écrite directement dans les classes modèles. Ces classes sont moins lisibles, certes, mais c'est plus confortable de tout faire dans un fichier. Je vous mets un petit exemple ici, il y en aura de plus conséquent en annexe.

Qui plus est, quand vous serez confronté à une erreur, la plupart des réponses que vous trouverez sur Internet seront dans ce format d'annotations, ce qui n'est pas négligeable.

```
/**
 * @var boolean
 *
 * @ORM\Column(name="isValide", type="boolean", nullable=false)
 */
private $isvalide;

/**
 * @var \Doctrine\Common\Collections\Collection
 *
 * @ORM\ManyToMany(targetEntity="Lambda\LambdaBundle\Entity\Categorie",
          inversedBy="items")
 * @ORM\JoinTable(name="liencategorie",
 *   joinColumns={
 *     @ORM\JoinColumn(name="idItem", referencedColumnName="idItem")
 *   },
 *   inverseJoinColumns={
 *     @ORM\JoinColumn(name="idCategorie", referencedColumnName="idCategorie")
 *   }
 * )
 */
private $categories;
```

Cela n'est pas très digeste, mais c'est ce que vous devrez écrire si votre base de données est vierge et que vous voulez que Doctrine crée automatiquement vos tables et vos clés étrangères.

Sans vous tromper ne serait-ce que d'une majuscule, ça va sans dire. Rassurez-vous, Doctrine a des systèmes de vérifications pour pallier aux problèmes d'intégrité.

En ce qui concerne l'attribut 'categories', on peut remarquer cette annotationLes collections Symfony sont très particulières :

```
@ORM\JoinTable(name="liencategorie"
```

C'est une relation Many-to-Many. Ça signifie que Doctrine gèrera automatiquement une table de lien, qui ne fera même pas partie de vos classes. Ça veut dire aussi que doctrine, pour cette attribut catégorie, créera non pas des chaînes de caractères ou des entiers, mais des 'Collections'. Les collections Symfony sont très particulières, et vous verrez qu'il est difficile de travailler avec ces objets (dans certains cas, et plus facile dans d'autres). En effet, si vous avez des requêtes SQL particulières à faire, SQL ne comprend pas les collections, et il est difficile de persister ces objets en base de données, puisqu'il est difficile d'accéder à un des attributs de la collection en particulier.

Avec ces objets précis, attendez-vous à passer beaucoup de temps sur l'écran d'erreur de Symfony lors de votre apprentissage, et faites de Google votre meilleur ami.

Les entités

Une entité dans Symfony est une classe modèle. De ce fait c'est aussi le nom d'un objet en mémoire qui est le reflet de cette classe modèle.

Par exemple, une entité User, contenue dans la variable $user, a été créée à partir de la classe modèle User. Les deux sont des entités.

Et ce sont ces objets que Symfony et Doctrine gèrent. Doctrine, notamment, se charge de lier votre couche métier et de persister vos données. Si vous aviez l'habitude d'écrire une requête SQL pour avoir l'identifiant d'un utilisateur, …. oubliez ça !! Avec Symfony, tout, absolument **tout**, est objet. Y compris dans les vues.

~~« SELECT * FROM User WHERE username = 'Toto' »~~

```
[Dans le controleur User]

$toto="Toto";
$user = $this->findUserByUsername($toto)->getId();
```

Là vous vous dites, Bon. De toute façon, la fonction « findUserByUsername() », il a fallu l'écrire, et dedans, il y a du SQL, et donc ça revient au même, sauf que ce n'est pas rangé au même endroit. Pas du tout :

- Le select SQL classique vous renverra un Array d'array (une ligne = un array de valeurs. Plusieurs lignes = un array de lignes). On accède aux valeurs en entrant la clé associée.
- Doctrine, elle vous renverra un objet [ou collection d'objets] User, calqué sur votre classe modèle. On ne va donc pas chercher la valeur associée à une clé, mais directement un attribut.

Les méthodes magiques Doctrines :

Même si cela n'est pas vraimen magique, ça fonctionne exactement comme les méthodes magiques PHP, surchargeables. Vous les connaissez : __CALL, __CONSTRUCT, __SET, __GET, etc.

Et bien Doctrine fait la même chose, en parcourant le nom des fonctions qu'on appelle, et en 'devinant' ce que l'on veut.

Doctrine dispose de beaucoup de ces fonctions très utiles, appelées 'fonctions magiques', qui vous feront gagner du temps passé en DAO.

En théorie, on peut presque tout faire avec ces seules fonctions. Je vous invite vivement à lire la documentation, qui est disponible ici :

http://docs.doctrine-project.org

Décryptons findUserByUsername() :

- Find : là, Doctrine sait qu'on veut trouver une entité. Elle retournera donc un objet PHP.
- User : On veut quoi ? Une entité User. Doctrine va donc aller chercher la classe modèle User, et retournera un objet User : ->$user .
- By : Doctrine sait maintenant qu'on ne cherche pas tous les User, mais un ou plusieurs Users bien particuliers. Qui plus est, elle attend maintenant un paramètre.
- Username : Désormais elle sait quoi chercher dans la base de données. Elle veut les Users dont l'attribut Username sera égal à celui qu'on lui fournira dans la parenthèse. Il doit être un attribut déclaré de la classe User. (n'espérez pas avec findUserByChaussuresvertes()).

Il convient toutefois d'être très carré avec ces méthodes. Elles ont des paramètres de sortie inaltérables. Par exemple une méthode findByQuelquechose() est prévue pour sortir plusieurs résultats, et ce, même s'il n'y en a qu'un. Cette méthode retourne donc dans tous les cas une **collection** de résultats. Si vous êtes sûr de n'avoir qu'un résultat à votre requête, utilisez plutôt les méthodes scalaires. Elles, sont prévues pour ne retourner qu'un seul résultat, qui sera donc plus facile à gérer (pas besoin de boucle). Elles s'écrivent très simplement : find**One**ByQuelquechose() .

Doctrine s'occupe donc de tout ce qui touche à la base de données : lire écrire modifier, chercher. Mais pour ça il faut lui en donner l'ordre.

Attardons-nous donc sur les contrôleurs, qui, eux, donnent les ordres.

Les contrôleurs

Ce qu'est un contrôleur dans Symfony

La théorie est très simple. Un contrôleur, dans Symfony, est une fonction, appelée par la vue (via un <button>, ou un <a>, bref un lien en général). Cette fonction doit obligatoirement envoyer une réponse, matérialisée par une vue, à laquelle on peut passer des paramètres.

Comme tout dans Symfony, ils sont très cadrés. Le contrôleur a un nom, celui que vous voulez, toujours suffixé de 'Controller'. La fonction, elle, doit obligatoirement s'appeler [quelquechose]Action. Par exemple :

```
public function connexionAction(){
        //votre logique
}
```

Ce couple Contrôleur/Action est interprété par Symfony comme une route. Par exemple, cette action est appelée par le routeur de cette façon :

```
login:
    path: /login
    defaults: { _controller: LambdaBundle:Security:connexion }
```

Depuis la vue, sur notre bouton, <a href="{{ path:'login' }}"> (ne tenez pas compte de la syntaxe, nous y reviendrons), le bouton va appeller la route 'login' du routeur. Cette route cible la fonction 'connexionAction' du contrôleur 'SecurityController'. En cliquant sur ce bouton, dans la barre d'adresse de votre navigateur, vous verrez l'adresse :

http://[racine du site]/login

D'ailleurs, si vous tapez cette adresse directement, le résultat est le même.

Mais rappelez-vous, un contrôleur doit **TOUJOURS** renvoyer une réponse.

Il y a plusieurs façons d'envoyer une réponse. La version longue, la version raccourcie, qui en général renvoie sur une vue, avec des paramètres. Il y a aussi la redirection, etc. Concentrons-nous sur le plus simple :

```
return $this->render('LambdaBundle:connexion.html.twig', array(
        'numero' => $numero
        ));
//Affiche la vue Connexion du Bundle Lambda, en passant la variable numéro.
```

Le return de la fonction renvoie la fameuse réponse, soit : la vue connexion.html.twig, avec comme paramètre la variable 'numéro', qui vaut $numero, qui a été calculée dans la fonction de notre contrôleur. Cette variable numéro pourra donc être appelée au besoin dans notre vue connexion.

Dans Symfony, qui suit le modèle MVC, tout est bien cadré. Ainsi, dans les contrôleurs, on peut utiliser des méthodes de récupération particulières.

Par exemple, pour récupérer l'utilisateur connecté à l'application dans le contrôleur, on utilise :

```
$user = $this->get('security.context')->getToken()->getUser();

$user->getUsername(); //rappelez-vous, tout est objet !
```

Pour avoir le nom de l'utilisateur connecté. Il est possible de passer cette variable à la vue de cette façon, mais il est aussi possible d'y accéder directement, puisque l'utilisateur connecté à l'application fait partie des variables superglobales. Ainsi, en tapant {{ app.user.username }}, nous y avons également accès.

Quel intérêt donc de chercher cet utilisateur dans le contrôleur ? Simple : app.user.username de la vue fonctionne si effectivement il y a un utilisateur connecté à l'application. Si l'utilisateur n'est pas authentifié, la variable ne sera pas définie, et son appel lèvera une erreur.

Ce code dans le contrôleur fait appel à l'équivalent Symfony de la variable de session PHP, en plus abouti. Il est toujours possible d'accéder (et de changer) la variable PHP : $_SESSION au besoin. Mais $_SESSION est un array(), pas un objet. La variable superglobale Symfony 'app', permet d'accéder à beaucoup d'information concernant l'application. On peut citer l'utilisateur, la requête courante, la session, l'environnement (production ou développement), et le débug.

Le meilleur ami du contrôleur : le repository

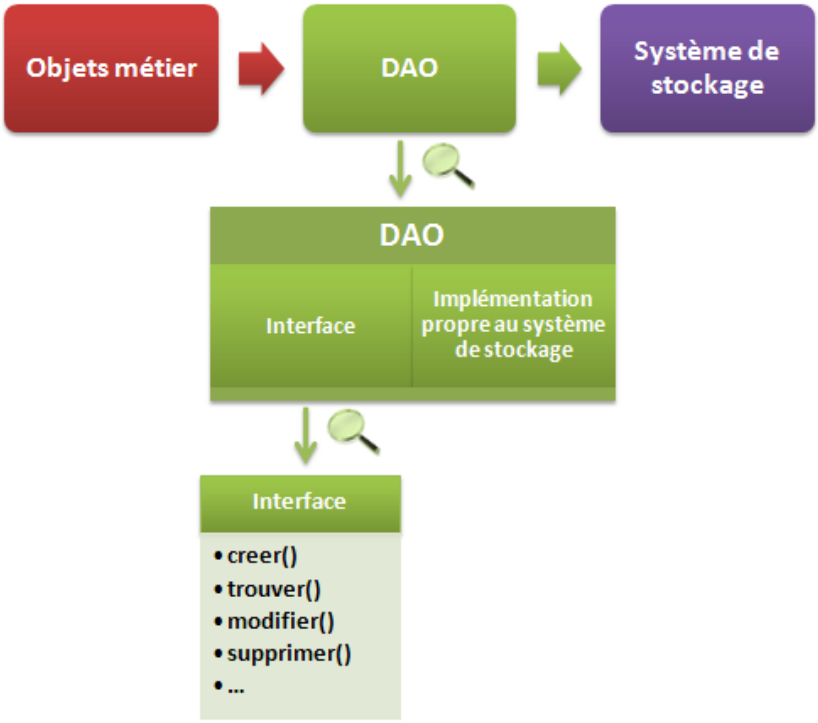
Un repository (traduction officielle : le 'dépôt'), est un endroit particulier, situé vraiment à mi-chemin entre les classes modèles, et les contrôleurs.

Dans les classes modèles, on ne fait objectivement aucun traitement de données, sauf s'il est nécessaire avant la création d'un objet. En tout cas il est réduit à son minimum. Les classes modèles ne servent qu'à définir des entités.

Un contrôleur, dans la philosophie Symfony, doit être le plus court possible aussi. Il doit, comme on l'a vu, renvoyer une réponse, sous quelques conditions.

Un repository permet donc justement de faire des requêtes et des traitements particuliers, sans alourdir ni les entités, ni les contrôleurs, dans le cas où il faut aller plus loin que les fonctions apportées par Doctrine.

C'est l'équivalent des classes de DAO, en somme.



Dans notre contrôleur, on va donc récupérer des objets, et des requêtes, les traiter sur place si c'est court, et si on a besoin de traitements particuliers, de requêtes particulières, sur un objet particulier, alors on utilisera les Repository.

J'ai bien dit 'objet particulier'. En effet un repository (qui n'est pas obligatoire, mais fortement recommandé), est affilié à une classe d'entité, et une seule. Par exemple un UserRepository est le dépôt [de fonctions] de la classe User. Il va donc traiter des objets User, et ses affiliations.

Déclaration d'un Repository

Un repository se déclare *via* les annotations Doctrine, le plus simplement du monde :

```
/**
 * User
 * @ORM\Entity(repositoryClass="Lambda\LambdaBundle\Repository\UserRepository")
 * @ORM\Table(name="user")
 * @UniqueEntity(fields={"username", "email"}, message="Ces valeurs sont utilisées !")
 *
 */
class User implements UserInterface, \Serializable {
```

L'écriture @ORM\Entity définit une entité, une classe concrète. On définit juste le Repository qu'on a créé dans les parenthèses qui suivent.

Utilisation dans un contrôleur : Simplement (quand on le sait), logiquement en tout cas.

Nous allons nous occuper d'entités, que ce soit dans les classes modèles, ou le repository qu'on va appeler. On va donc appeler Doctrine. On va ensuite appeler une fonction chargée de gérer (manager) les entités. Et on va lui indiquer quelle classe (ou repository) utiliser. Faisons cela dans un contrôleur divers, parce que les actions de sécurité liées aux connexions sont un peu particulières.

```
public function afficherAction(){
        //$em signifie : EntityManager
        $em = $this -> getDotrine() -> getManager() ;
        $items = $em -> getRepository('LambdaBundle : User') -> findAll() ;
}
```

Simple, quand on le sait. Une fois que vous avez écrit :

```
$em -> getRepository('LambdaBundle : User') ;
```

Vous avez la maîtrise des utilisateurs.

Toutefois, vous noterez que 'User' n'est PAS un repository. En fait, Symfony ne fait aucune distinction entre les modèles et les repository.

Si Doctrine dispose de la fonction que vous recherchez (ici : `findAll()`), elle l'éxécute.

Si Doctrine ne connait pas la fonction que vous appelez, elle ira la chercher dans le repository, déclaré en haut de la classe User.

Si cette fonction n'est pas déclarée dans le repository, Doctrine vous le signifiera par une page d'erreur : fonction non trouvée.

Justement. Pourquoi un repository si grâce aux fonctions 'magiques' on récupère tout ? Par souci de performance, et surtout de flexibilité !

Le DQL

Vous connaissez le langage SQL, qui permet d'interroger des bases de données ? Celui-ci en est un autre.

Le DQL existe pour pallier aux limitations du SQL. Quand vous interrogez une base de données en SQL, votre requête vous retourne un array de lignes, qui elles-mêmes sont un array de …. Tout et n'importe quoi. Des chiffres dans le cas d'un count, des int avec des strings, etc.

Le DQL est différent. C'est le langage d'interrogation de bases de données de Doctrine. Et c'est le langage que vous devrez utiliser dans vos Repository, pour les éventuelles requêtes particulières que vous aurez à faire.

Les avantages : le DQL est un langage très proche du SQL, vous ne serez pas dépaysés. Il est assez intuitif, et permet de faire énormément de choses, mais en objet !

Construction : Il s'appelle avec le CreateQueryBuilder de Symfony. Le select est toujours un Alias de la classe de laquelle il est appelé. Le UserRepository ?

=> createQueryBuilder('u').

- ⇨ Renvoie un [ou plusieurs] objets User.

Exemple :

```php
<?php
// Dans un repository

public function myFindOne($id)
{
  $qb = $this->createQueryBuilder('a');

  $qb
    ->where('a.id = :id')
    ->setParameter('id', $id)
  ;

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

Vous avez deviné un de ses avantages. On peut lui passer des paramètres. En SQL aussi vous me direz, mais ici c'est simple.

Les changements de syntaxe par rapport au SQL. En fait ils sont peu nombreux. Le plus perturbant est sans doute le JOIN (LEFT, RIGHT, INNER).

On a tous l'habitude d'écrire :

'SELECT Chose FROM TableA LEFT JOIN TableB ON Condition'

Mais en DQL le ON n'existe pas. Il se dit WITH.

J'avoue qu'au début c'est agaçant.

Le gros avantage du DQL, c'est sa modularité.

Remarquez sur l'image suivante que l'on n'a pas utilisé 'createQuerybuilder'. Vous pouvez remarquer que la fonction prend un paramètre. Un paramètre que l'on connait …

```
12    public function whereCurrentYear(QueryBuilder $qb)
13    {
14      $qb
15        ->andWhere('a.date BETWEEN :start AND :end')
16        ->setParameter('start', new \Datetime(date('Y').'-01-01'))  // Date entre le 1er
    janvier de cette année
17        ->setParameter('end',   new \Datetime(date('Y').'-12-31'))  // Et le 31 décembre
    de cette année
18      ;
19    }
20  }
```

Je vous ai dit que le DQL était modulable…. Et c'est vrai. Qui n'a jamais rêvé de rajouter des conditions à volonté à une requête ? De la changer à volonté, selon le bon vouloir du contrôleur ???

La réponse, justement, se trouve dans le contrôleur :

```
1  <?php
2  // Depuis un repository
3
4  public function myFind()
5  {
6    $qb = $this->createQueryBuilder('a');
7
8    // On peut ajouter ce qu'on veut avant
9    $qb
10     ->where('a.author = :author')
11     ->setParameter('author', 'Marine')
12   ;
13
14   // On applique notre condition sur le QueryBuilder
15   $this->whereCurrentYear($qb);
16
17   // On peut ajouter ce qu'on veut après
18   $qb->orderBy('a.date', 'DESC');
19
20   return $qb
21     ->getQuery()
22     ->getResult()
23   ;
24 }
```

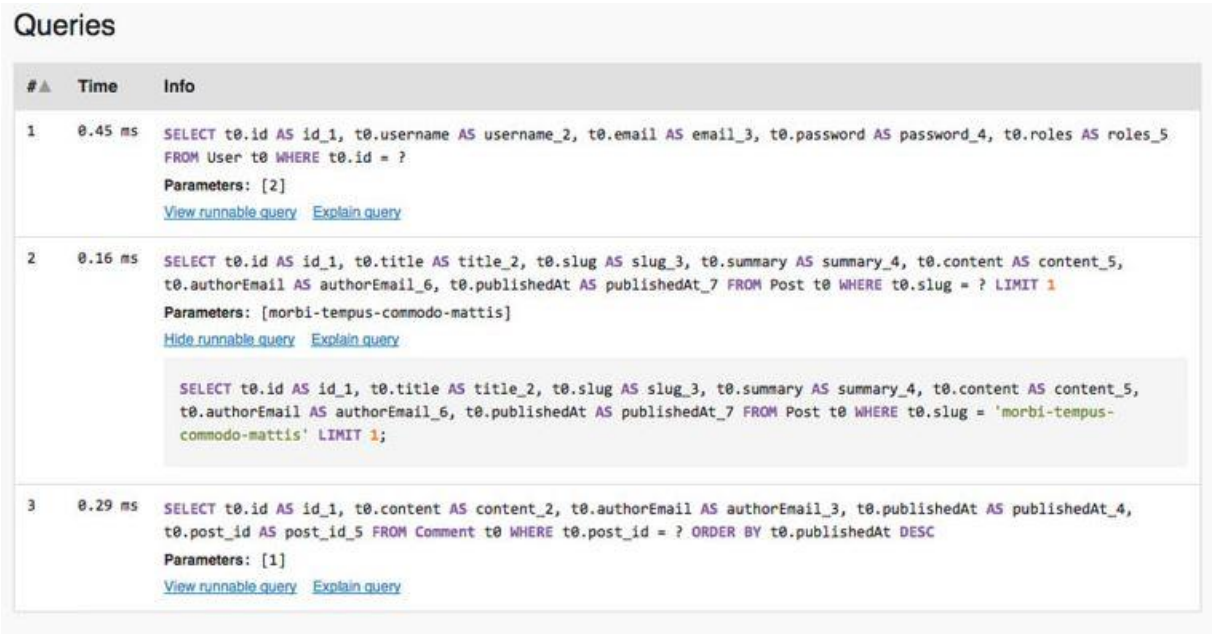
Cela n'est pas si facile à faire en SQL, alors qu'avec le DQL, c'est très simple. On peut construire un repository rempli de clauses WHERE dans des fonctions, appelables à volonté, et au besoin.

Du coup, on peut rajouter des clauses WHERE à la volée, des AND, tout ce qu'on veut, où on veut, quand on veut. Imaginez une requête qui cherche les messages dont le destinataire est un utilisateur particulier. On l'écrit, et ensuite on peut se faire une fonction qui rajoutera des conditions de recherche via des AND, rajoutera des LIMIT, de l'ordonnancement dans les réponses…

Je vous ai parlé de performances, sans trop m'y attarder. En effet, Doctrine a tendance à récupérer toutes les informations d'une entité, y compris les différentes associations entre classes modèles.

Par exemple, un utilisateur dispose d'une ou plusieurs adresses, disons un pool d'adresses, dans une relation ManyToMany déclarée dans la classe User (n-a-n, ou *--* dans un schéma de base de données). Et bien Doctrine va aller chercher, à chaque appel de la classe User, le pool d'adresses associé. Imaginons que nous avons des millions d'utilisateurs dans notre application, et que chacun dispose de plusieurs adresses.

Bien sûr Doctrine fait ses requêtes SQL de façon transparente pour l'utilisateur, et même pour nous, mais leur nombre peut devenir colossal. On peut les visualiser grâce au Profiler Symfony, cette barre visible en environnement de développement.



À tel point que pour contrôler l'appétit insatiable de Doctrine, et pour améliorer les performances, certains ne définissent pas du tout de relations entre les classes, pas du tout de clés étrangères sur les tables, et font exclusivement des requêtes dans les Repository, au besoin.

Les contrôleurs et Doctrine

Vous vous souvenez de la fonction `'ajouterAction'` de tout à l'heure ?

Faisons maintenant une fonction `'nouveauAction'`, permettant de créer un nouveau groupe, du contrôleur gérant des groupes.

Dans ce contrôleur, on va gérer la création d'un nouveau groupe, par un utilisateur, qui sera, en toute logique, l'utilisateur qui a appuyé sur le bouton de création de groupe, et qui donc est l'utilisateur connecté à l'application.

Voyons un peu, avant toute chose, l'entité Groupe.

L'entité Groupe a une relation Many-To-Manynous décidons avec l'entité User. Cela signifie qu'un utilisateur peut appartenir à plusieurs groupes, et qu'un groupe peut contenir plusieurs utilisateurs.

Dans l'entité groupe, on aura donc un attribut « Users » qui sera une collection d'utilisateurs (c'est ce que Doctrine vous renverra, et c'est plus simple à comprendre au pluriel) ;

Et dans l'entité User, on aura un attribut « groupes » qui sera une collection de groupes.

Il y a une entité propriétaire, et une entité inverse. Dans une relation Many-To-Many, peu importe le propriétaire, mais par logique, nous décidons que le propriétaire des groupes sera l'entité Groupe.

Donc, dans l'entité groupe :

```
/**
 * @var \Doctrine\Common\Collections\Collection
 *
 * @ORM\ManyToMany(targetEntity="Lambda\LambdaBundle\Entity\User", inversedBy="groupes")
 */
private $users;
```

Et dans l'entité User :

```
/**
 * @var \Doctrine\Common\Collections\Collection
 *
 * @ORM\ManyToMany(targetEntity="Lambda\LambdaBundle\Entity\Groupe", mappedBy="users")
 */
private $groupes;
```

Très peu de différences n'est-ce-pas ? En fait il n'y en a qu'une. L'entité propriétaire comporte la notation `'inversedBy'`, et l'entité inverse possède `'mappedBy'`.

Je vous fais la grâce de la table de lien et des jointures de colonnes qui sont en fait optionnelles, Doctrine gérant tout. Ma base de données était déjà créée, et donc j'ai dû les mettre.

Les getters et les setters

S'agissant de collections, ils sont un peu particuliers. On a bien sur les getUsers() et setUsers() dans l'entité User, et les mêmes dans l'entité Groupe ;

Mais on a aussi, comme ce sont des collections, les addElement($groupe), et removeElement($groupe) côté User, et vice-versa.

Ces éléments un peu particuliers, vous l'aurez deviné, permettent de rajouter un élément groupe à une collection de groupes déjà existante.

Le contrôleur

Finalement nous y sommes. Le contrôleur. Ce chapitre s'intitule 'Les contrôleurs', et on n'y arrive que maintenant. Mais tout est tellement interdépendant, qu'il fallait en passer par là.

Dans ce contrôleur, qu'est-ce qu'on veut… On veut créer un nouveau groupe, avec tous ses attributs (un nom, une description). C'est un bon début. Vous verrez que l'air de rien, il y a plus d'attributs que l'on croit.

Désolé pour les commentaires, il y en a partout. Mais bon, il faut bien comprendre…

```
    public function newAction(Request $request, UserInterface $user)
    {
        $officier = $this->getUser();
//Et oui, il faut bien administrer le groupe !! Du coup, le créateur du
//groupe est automatiquement officier !
//$this->getUser() est un raccourci Symfony pour avoir l'objet
// utilisateur connecté à l'application.
        $groupe = new Groupe();
//Oh un nouveau groupe. Etrange…

        $form   =   $this->createForm('Lambda\LambdaBundle\Form\GroupeType',
             $groupe);
//formulaire : demande nom et description (créé dans GroupeType.php, auquel
//on envoie notre objet Groupe)

        $form->handleRequest($request);
//Réception / traitement du formulaire

        if ($form->isSubmitted() && $form->isValid()) {
            $em = $this->getDoctrine()->getManager(); //ça on connait

            if ($user != null) { //s'il y a un user, sinon rien

            $groupe->addUser($user);
//le créateur devient membre du groupe
            $groupe->addOfficier($officier);
//et officier aussi

            $user->addGroupe($groupe);
//et bien sûr, on ajoute le groupe à
//la collection de groupes de l'utilisateur

            $em->persist($groupe); //objet groupe préparé à être écrit
            $em->persist($user); //idem pour user: prêt à l'entrée en base
            $em->flush(); //ordre à Doctrine : écrit en base !!
      }
            Return $this->redirectToRoute('base_groupe_index', array(
                                'groupe' => $groupe
            )); //affiche le chemin 'base_groupe_index',
                //       avec pour paramêtre :'groupe' égal à $groupe
```

```
        }
```

Très peu de lignes, mais pour faire de grandes choses. On a :

- Récupéré un objet User,
- Créé un nouveau groupe,
- On lui a donné un nom et une description (Symfony s'en charge *via* le formulaire),
- On a ajouté le créateur du groupe en tant que membre du groupe,
- Également en tant qu'officier (pouvant administrer le groupe. Virer des gens, etc.),
- On a ajouté le groupe dans la collection de groupes de l'utilisateur (peu importe qu'il y en ait zéro, un, deux, 100, on l'ajoute). Dans la méthode addGroupe() de l'entité User, toutefois, on a spécifié que ce groupe ne devait pas déjà exister,
- On a persisté ces deux objets (rendus prêts à écrire par Doctrine, plus aucune modification dessus donc),
- Et enfin, Doctrine a fait les requêtes (y compris la table de liens, n'oubliez pas) de façon automatique.

Vous pouvez voir ces requêtes dans les logs, mais la meilleure façon c'est encore d'aller voir votre base de Données. Et on a bien un groupe de créé dans la table des groupes, avec son nom et sa description, un membre, un officier, mais pas de colonnes 'users' dans la table groupe, et dans User, rien …..

Comment ça rien ?

Il n'y a même pas de colonne 'groupes' dans la table User en base de Données. Il n'y en a jamais eu d'ailleurs.

Vous vous souvenez de cette histoire de table de liens, gérée automatiquement par Doctrine ?

Hé oui !! Magnifique, non ??? Doctrine a tout fait. Comme souvent avec Symfony, si ça ne fonctionne pas avec ce code épuré > regardez du côté des entités, les classes modèles.

Si les classes modèles tiennent la route, Doctrine les gèrera simplement et efficacement.

Les relations Many-To-Many sont réputées être les plus difficile à gérer. Et ça ne nous a pris que quelques lignes.

TWIG : Les Vues

Twig ?

Quand on ne sait pas ce que c'est, on se demande à quoi cela peut servir. En fait, c'est votre nouveau meilleur ami. Si si, vraiment.

Du coup, qu'est-ce que Twig ? C'est un moteur de templates (la traduction officielle est 'modèle', ce qui, dans notre cas, serait problématique) pour PHP.

À quoi ça sert et qu'est-ce qu'on fait avec. Comme ça définition l'indique, on peut faire des pages qui ne seront que des templates pour les pages définitives.

- Par exemple, on peut envoyer des emails qui auront la même structure, on ne changera que les données. On verra un exemple par la suite.
- La seconde grande innovation de Twig, c'est l'héritage. Avec Twig, une page peut hériter d'une parente, qui hérite d'une parente, qui hérite d'une parente.
- La troisième innovation, c'est bien sûr la syntaxe. C'est simple, fluide et logique. Et vous verrez que ça ressemble beaucoup à l'héritage, d'une certaine façon.

L'héritage

Hé oui, je vais traiter l'héritage avant la structure. En fait, twig n'a que peu d'intérêt sans l'héritage.

Et c'est très simple : une page peut hériter d'une autre page. Par exemple, une page de résultats des calculs d'un contrôleur peut hériter d'une page de base. Dans la page de résultats, on ne mets donc que les résultats, alors que dans la page de base, on met le Doctype, le <HEAD> avec l'appel des feuilles de style, l'appel des scripts etc etc.

Et ça y est, vous êtes débarrassé de tout ça. Si toutes vos pages héritent de votre page de base, vous ne l'écrivez qu'une fois. Ce n'est qu'une question de choix.

Par exemple, dans le projet Lambda, ma page de base inclut un header et un footer. Normal en somme. Donc si je fais hériter toutes mes pages de la page de base, elles auront toutes un header et un footer.

Mais je vais créer un second template, de la même page de base, avec les css et le javascript, mais sans header ni footer, pour des éventuelles pages 'popup'.

En fait, avec twig, quand on crée une page, il faut toujours réfléchir à ce qui change par rapport à une autre, et ce qui peut être mis en commun. S'il y a quelque chose en commun, il faut faire hériter !

On peut parfaitement créer une page, qui hérite d'une autre page, qui hérite d'une autre page, qui hérite enfin de la page de base.

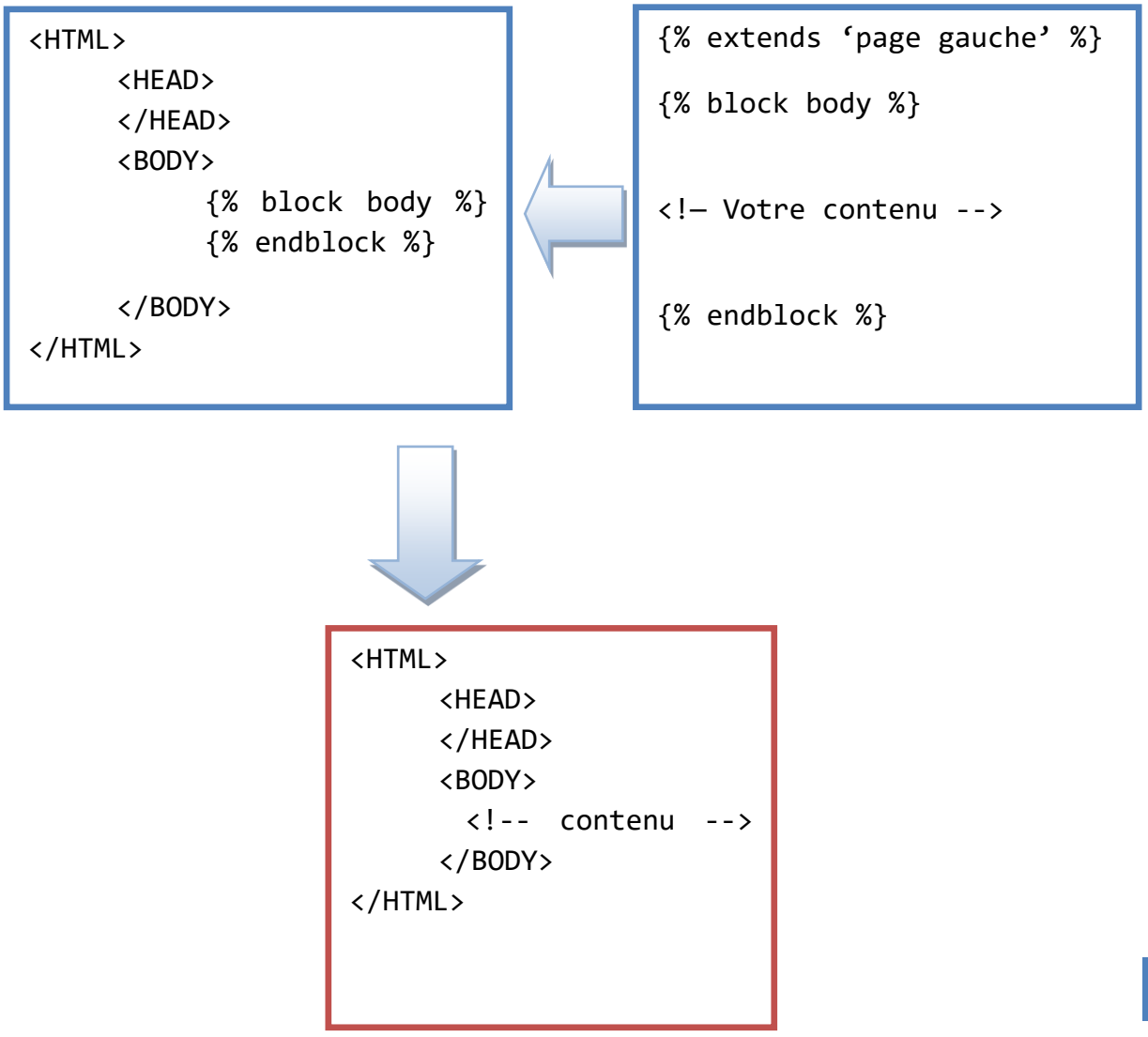
Vous comprendrez mieux avec les blocs…

La structure :

Là encore c'est simple, on fait ce qu'on veut, enfin presque…

Tout n'est qu'une question de blocs. Les blocs doivent être définis autant sur la page qui hérite que sur la page héritée.

Par exemple :



Simple, n'est-ce pas ? Et on peut mettre les blocs où on veut ; on pourrait même modifier le Doctype à la volée, pourquoi pas (cela n'aurait que peu d'intérêt, mais ça n'est qu'un exemple).

Prenons un autre exemple, très concret : la balise titre d'une page HTML.

Cette balise, placée dans la balise <HEAD> permet d'afficher le titre de la page dans la fenêtre ou l'onglet du navigateur du client. Imaginons que sur la page de base je la définisse comme ceci :

<TITLE>{% block titre %}{% endblock %} – NomdemonSITE</TITLE>

Les pages qui héritent de cette page …. Héritent de cette page. Ça ne change rien.

Mais s'il me vient l'idée lumineuse de mettre, n'importe où dans ma page qui va hériter, ceci (en haut serait judicieux) :

{% block titre %}NomdemaPAGE{% endblock %}

Au final, la balise <TITLE> sera rendue de cette façon pour l'utilisateur :

<TITLE>NomdemaPAGE – NomdemonSITE</TITLE>

La syntaxe

Je commence beaucoup de paragraphes sur Twig en disant que c'est simple. Hé bien ce sera encore le cas. Il n'y a que deux règles absolues à respecter. C'est tout …

- Règle 1 : Les variables se mettent entre doubles accolades, comme ça: {{ variable }}. Pas de '$' comme en php, rien de tout ça…
- Règle 2 : Les fonctions (celles de Twig) s'écrivent avec accolade-pourcent, comme cela : {% if condition %}chose{% else %}autrechose{% endif %}

Petit truc : personnellement je code sous NetBeans. Je me suis fait des macros qui tapent directement {{ }}, et {% %}, ce qui me fait gagner un temps précieux.

Oui oui, il y a aussi des if, des for en twig. Admettons que le contrôleur nous envoie comme variables une liste (un array) de choses, disons $lesobjets.

Hé bien comme en PHP, on peut boucler dessus.

Et pour faire cela, Twig est très intuitif, et c'est son grand avantage. Voyons l'exemple d'une boucle for.

Admettons que nous rendons une vue, appelée bien sûr par un contrôleur, et que ce contrôleur ait passé comme variable à cette vue une liste d'objets appelée 'lesobjets'. On veut, dans notre vue, pour construire une table, boucler sur cette liste d'objets, et récupérer l'attribut 'nom' de chaque objet.

For, en twig, s'écrit :

```
{% for unobjet in lesobjets %}   // Comprenez : 'pour un objet parmi lesobjets'
        {{ unobjet.nom }}        // 'affiche l'attribut 'nom' de l'objet unobjet'
{% endfor %}                     // 'Fin de la boucle, passe à la suite'
```

[À la place de 'unobjet', vous pouvez en fait mettre ce que vous voulez, ça n'a pas d'importance. Pour la lisibilité de votre code toutefois, il vaut mieux mettre quelque chose d'explicite. Quand j'écris « {% for unobjet in lesobjets %} », même deux ans plus tard je saurais ce que cela veut dire.]

Facile !! Twig s'écrit comme on pense. Et c'est un gain de temps considérable. Bien sûr, dans le cas d'une table, on peut parfaitement mettre des <tr><td> </td></tr> autour de notre variable, histoire que tout ça serve à quelque chose.

Les filtres

Des filtres … oui oui des filtres. Il y en a plein, et je vous invite à lire la documentation pour tous les connaitre. Ce sont des fonctions particulières qui se greffent à des variables (voir les changent) pour être 'visualisables' plus facilement.

L'exemple classique : imaginons que nous avons un utilisateur enregistré, récupéré en base de données. Cet [objet] utilisateur possède un attribut $membredepuis.

Le contrôleur nous a envoyé l'objet 'utilisateur' (avec tous ses attributs, y compris donc 'membredepuis' => pensez objet).

Dans Twig, on va appeler cet objet, demander l'attribut qui nous intéresse, puis placer un filtre dessus pour l'afficher de façon « digeste » pour nous humains :

```
{{ utilisateur.membredepuis|date('d-m-Y') }}
```

Il va donc considérer l'attribut membredepuis de l'objet utilisateur comme un date, et la mettre au format 'jour-mois-annéecomplète'.

[D'ailleurs, pour votre information, en ce qui concerne les dates, le filtre est obligatoire dans Twig. Si vous ne le mettez pas, vous aurez une erreur vous disant qu'il ne peut pas convertir un objet 'Date' en chaîne de caractères.]

Je vous avais dit que les variables, dans Twig, fonctionnaient comme l'héritage. Voyons cela avec les variables globales :

```
{{ app.user.username }}
```

Signifie dans Twig : je veux le nom de l'utilisateur connecté à l'application.

Rappelez-vous, utiliser ça sur une page où aucun utilisateur n'est connecté est problématique … et c'est logique. Mais si j'écris (je filtre) :

```
{{ app.user.username|default('Non connecté') }}
```

Twig va aller chercher le nom de l'utilisateur de l'application, qui sera par défaut 'Non connecté'. Autrement dit ce nom sera 'Non connecté' par défaut, et s'il en trouve un il le remplace par le vrai.

Pas besoin de if, de else, rien de tout ça.

Pour résumer, Twig est vraiment intuitif, on écrit les vues comme on pense. Le tout est de penser bien. Voyons ça avec un exemple très concret (oui, encore un, mais c'est celui que je vous promets depuis le début du chapitre).

Exemple concret : Personnalisation des pages d'erreur

Je vous l'ai dit : avant de se lancer, il n'y a qu'une question à se poser : qu'est-ce qui change, et surtout : qu'est-ce qui ne change pas…

Des pages d'erreur serveur, il y en a plein (pas loin de 40 !), mais on ne va s'occuper que des 5 principales. Comme ce n'est pas un site de e-commerce -> pas de paiement, on va éluder l'erreur 402.

Nous avons donc les erreurs :

- 400 : mauvaise requête http
- 401 : non autorisé (sur le serveur)
- 403 : interdit (par l'application)
- 404 : page non trouvée
- 500 : erreur interne du serveur

Disons que nous voulons une page toute simple, de ce genre-là :

Je vous repose donc la question qu'il faut obligatoirement se poser :

Qu'est-ce qui change … ou plutôt qu'est-ce qui reste ?

Facile : TOUT reste, ou presque. Les seules choses qui changent, c'est le texte [sans mise en forme, du texte brut], et peut-être l'image, mais c'est à discuter.

Voilà la façon de procéder, du moins celle qui semble la plus simple :

- On crée une page erreur.html.twig, qui hérite de la page de base (autrement dit on hérite des CSS, du header, du footer, javascript et j'en passe).
- La page d'erreur ne contient que très peu de <DIV>, et une balise <IMG src='{% block adresseimgerreur %}{% endblock %}' />
- Parmi les div, un est comme ceci :Vousr <DIV>{% block texteerreur %} {% endblock %}</DIV>.
- On met en place sur cette page les autres <DIV>, les 'quotes', on règle nos styles ici, etc.
- Et on crée les 5 vraies pages d'erreur, qui hériteront de la page erreur.html.twig (qui elle-même hérite de la page de base, rappelez-vous).

Que contiennent donc ces vraies pages d'erreur ?

= > Rien !!!

Enfin si, l'adresse de l'image dans un bloc 'adresseimgerreur', et le texte de l'erreur dans un bloc 'texteerreur'.

Plein de raisons à cela : facile à maintenir, facile à trouver, facile à comprendre, les 5 pages d'erreur faite en 4 copier/coller, et en changeant le texte et une adresse.

Parce qu'au final, il n'y a que ça qui change.

De la même façon, si on veut modifier nos pages d'erreur, disons modifier la couleur du div de fond, on ne le fait que sur la page erreur.html.twig, et ça modifie les 5 !!!

Bien entendu

Je suis sûr que cette fois, vous avez compris l'intérêt de Twig…

Vous retrouverez la vraie page d'erreur, en totalité, dans les annexes.

Les services

Comme je ne suis pas un grand spécialiste de ce concept, je ne m'y attarderait que brièvement ici, histoire que vous compreniez qu'avec Symfony, c'est possible aussi !

L'architecture orientée service

L'objectif d'une architecture orientée services (SOA) est donc de décomposer une fonctionnalité en un ensemble de fonctions basiques, appelées **services**, fournies par des composants et de décrire finement le schéma d'interaction entre ces services.

La SOA et Symfony

Symfony intègre un moteur de services, appelé le 'Container'. Cette intégration vous permet, à vous, au besoin, de déclarer tous les services que vous voulez dans votre application Symfony, et de les appeler à volonté.

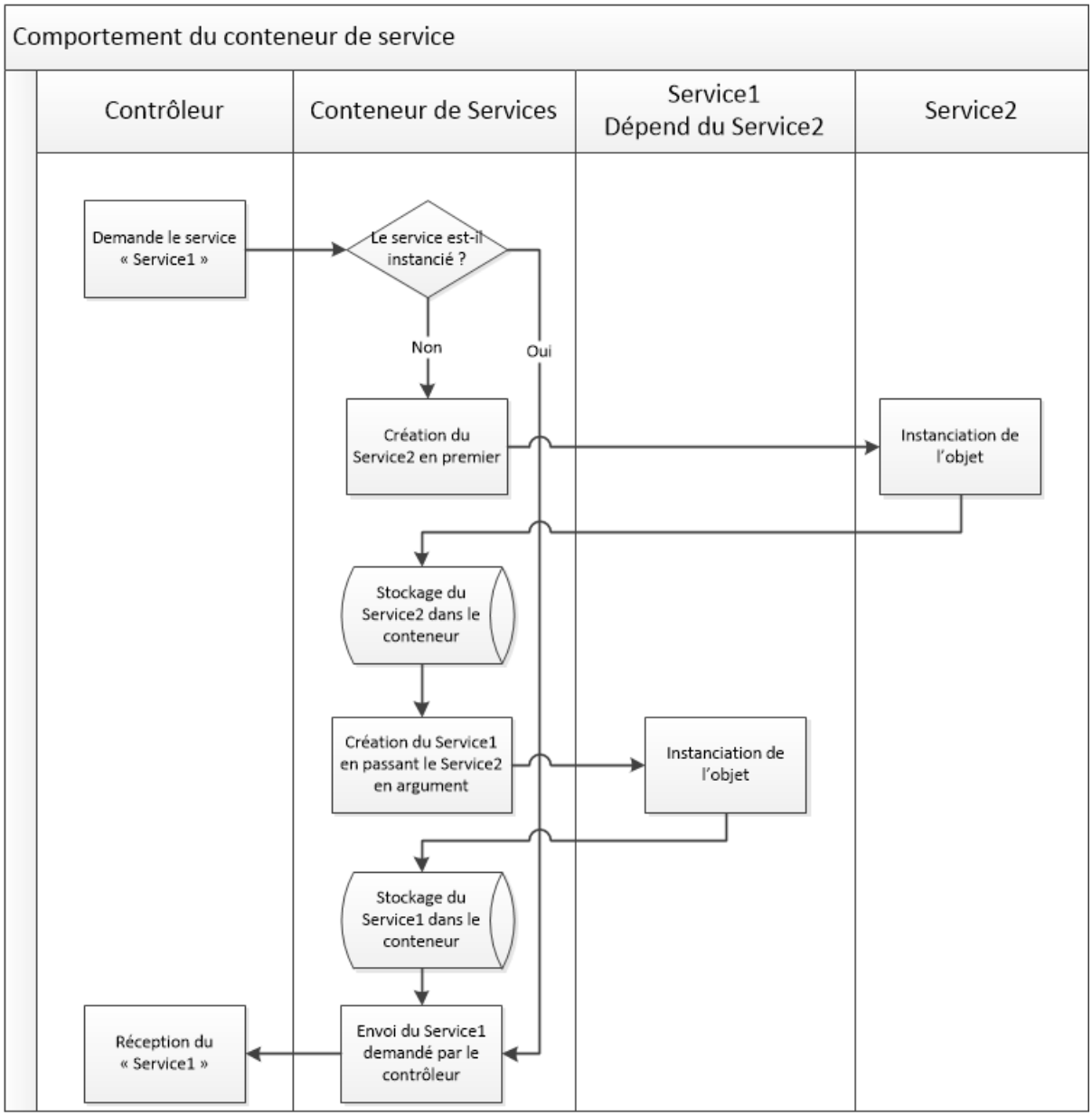
Un service est simplement un objet PHP, qui éxécute une fonction, tout comme plein d'autres objets PHP. Mais si on besoin des fonctions de cet objet très souvent [ou un peu partout dans notre application], il peut être intéressant de le déclarer en tant que service.

Il y a des services déjà existants dans Symfony bien entendu, qui sont appelé au besoin.

L'intérêt des services dans Symfony réside justement dans le Conteneur de service. La figure suivante vous fera comprendre à quel point il va vous éviter un casse-tête.

Fonctionnement du 'conteneur de services' :

La figure suivante montre le rôle du conteneur de services et son utilisation. L'exemple est constitué de deux services, sachant que le `Service1` nécessite le `Service2` pour fonctionner, il faut donc qu'il soit instancié après celui-ci. Et le conteneur va gérer ça automatiquement, de façon transparente.



Source: OpenClassrooms.com

Vous l'avez compris, ces services, mêmes s'ils sont accessibles de partout, ne sont instanciés qu'une seule fois. Ce qui vaut mieux que d'appeler une fonction à chaque changement de page par exemple.

Service plus concret

Prenons un exemple concret, concernant un service pour Twig, que nous avons mis en place par rapport à un besoin précis, et une lacune de Symfony.

Dans notre base de données, nous avons une table adresse, qui stocke les adresses de nos utilisateurs. Jusque-là, tout est normal. Symfony, pour ses formulaires, intègre tout un tas de champs préfabriqués, en rapport avec leur fonction.

Dans notre formulaire d'adresse, nous avons donc un champ 'CountryType', qui est un sélecteur de pays, sous la forme d'une liste déroulante, déjà peuplée de tous les pays du monde. En gros, c'est un champ de sélection de pays prêt-à-l'emploi.

L'ennui (pas très gros, je vous rassure), c'est que l'on n'a pas la main sur ce type de champ. L'utilisateur choisit un pays dans la liste, et après la validation du formulaire, Symfony peuple la table adresse, et surtout l'attribut 'Pays' de => « FR » pour France.

Jusque-là, rien de bloquant ou de gênant. Toutefois, au moment de l'affichage sur une vue, et après avoir parcouru la documentation de Twig, on se rend compte qu'il n'y a pas de filtre pour le rendu du nom de pays complet. On va donc rester avec 'FR' dans nos vues, ce qui ne nous convient pas du tout.

Bien entendu, on peut faire un switch / case dans un contrôleur pour corriger ça. Mais ça ne serait pas très propre et surtout ça serait laborieux. Qui plus est, si notre base de données est déjà peuplée, ça serait problématique.

On va donc déclarer un service !!!

On va donc se créer un classe d'extension de Twig. On va se servir du composant Intl de Symfony, qui gère tout ce qui a trait aux 'locales'.

Voilà cette classe :

```
use Symfony\Component\Intl\Intl;

class CountryExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return array(
            new \Twig_SimpleFilter('countryName', array($this,
'countryName')),
        );
    }

    public function countryName($countryCode){
        return Intl::getRegionBundle()->getCountryName($countryCode);
    }

    public function getName()
    {
        return 'country_extension';
    }
}
```

On s'est donc créé une nouvelle extension de Twig, en se servant du composant Intl.

Mais Twig ne sait pas que cette fonction existe, il ne la connait pas, elle n'est pas sur la liste des invités.

On va donc ajouter cette extension dans la liste des services, en modifiant le fichier 'services.yml' de l'application, pour y ajouter ceci :

```
services:
    lambda.twig.lambda_extension:
        class: Lambda\LambdaBundle\Twig\CountryExtension
        tags:
            - { name: twig.extension }
```

Maintenant, Twig connait la classe 'CountryExtension' que nous avons écrite plus haut. Si vous la regardez de plus près, vous verrez qu'elle contient une fonction 'countryName' qui prend en paramètre un 'countryCode', qui est en fait, le fameux FR ou EN, etc.

Et … c'est tout. De fait, maintenant on peut utiliser la fonction 'countryName' de partout. Enfin, comme c'est une extension Twig, partout dans nos vues. En fait on vient de rajouter un filtre Twig.

Maintenant, il suffit d'écrire, n'importe où dans nos vues :

```
{{ adresse.pays|countryName }} //l'attribut 'pays' de l'objet 'adresse', avec le
                               //filtre 'countryName'
```

Et le tour est joué, maintenant il nous affiche bien 'France' , à partir de adresse.pays, qui renvoie 'FR'.

Le conteneur de services de Symfony est vraiment puissant et facile. Il y a déjà un tas de services déclarés avec Symfony, qu'on peut utiliser de partout. Ça va de Doctrine, à un service de mail. On les utilise sans même s'en rendre compte, et ça rajoute à la facilité de coder avec Symfony.

Les Formulaires

Les formulaires sont très importants dans une application. Ce sont eux qui dialoguent avec l'utilisateur, et lui permettent d'entrer vraiment dans l'application. Ces formulaires leur permettent de s'authentifier, d'entrer des données, d'écrire des commentaires, de discuter avec d'autres utilisateurs, voir avec l'administrateur. Dans tous les cas ils ne sont pas à négliger.

Les formulaires de Symfony

Les formulaires de Symfony sont très particuliers. Ils sont toujours appelés d'une fonction d'un contrôleur, et c'est cette même fonction qui appelle le rendu du formulaire lui-même, qui attend qu'ils soient validés (nous reviendrons sur les contraintes de validation), et qui renvoie la vue à afficher une fois l'action terminée.

Exemple, la création d'un groupe :

Je crée le formulaire, j'affiche la vue du formulaire, et une fois le formulaire validé et le groupe créé, j'affiche le listing de tous les groupes.

Les champs automatiques

Symfony intègre beaucoup de champs de formulaires, chacun avec sa fonction précise. Je ne listerais ici que les champs les plus utilisés, mais je vous recommande de lire la documentation, car il y a forcément un champ qui existe pour votre besoin.

Il y a le TextType, le plus classique, le IntegerType pour les nombres, le PasswordType, qui est le champ à masquage de caractères, le CountryType dont on a déjà parlé plus tôt, et un champ un peu particulier :

EntityType, qui est un champ entité. Ce champ entité n'est pas un champ que Symfony lie à une de vos entités, mais c'est un champ que **vous** relierez à une de vos entités. C'est une extension du champ de type 'ChoiceType', dont le rendu est une liste déroulante. Et vous pouvez peupler cette liste avec des objets de vos entités, selon vos critères de choix.

Par exemple, cette liste peut contenir l'ensemble des catégories. Mais vous pouvez très bien décider de limiter ce choix à 5 catégories, de classer ces choix par ordre

alphabétique. Vous pouvez même peupler cette liste avec une requête sur la base de données, directement.

Vos propres champs

Un peu comme pour les contrôleurs, chaque champ a une convention de nommage propre : leur nom doit être suffixé de 'Type'. C'est important car vous pouvez également créer les vôtres, affiliés à vos entités. Par exemple si j'appelle dans mon contrôleur le formulaire 'GroupeType', ce sera la collection de champs, que Symfony considèrera comme un seul champ, qui sera directement lié à mon entité 'Groupe'.

Ainsi, lors de la construction de la classe 'GroupeType', on ne pourra ajouter de champs que sur des attributs de la classe 'Groupe'.

L'avantage est que vous avez le choix des champs à implémenter dans votre formulaire, ou non. Notre classe groupe comporte par exemple un attribut 'dateajout', qui est hydraté de la date actuelle à la création de l'objet Groupe. Il n'est donc pas utile de le faire figurer dans le formulaire.

```
class GroupeType extends AbstractType
{
    /**
     * {@inheritdoc}
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('nomgroupe',TextType::class, array('label' => 'Nom du groupe :'))
                ->add('description',TextareaType::class, array('label' => 'Description du groupe :'))
                ;
    }

    /**
     * {@inheritdoc}
     */
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Lambda\LambdaBundle\Entity\Groupe',

        ));
    }
```

Ici, nous n'avons ajouté que 'nomgroupe' qui représente le nom du groupe à créer, ainsi que la description du groupe. Tous deux sont des attributs de la classe Groupe.

Dans le contrôleur, on appelle notre formulaire simplement, de cette façon :

```
$groupe = new Groupe();
$form = $this->createForm('Lambda\LambdaBundle\Form\GroupeType', $groupe);
```

On crée un objet Groupe, et on le passe en paramètre au formulaire. Une fois le formulaire rempli par l'utilisateur, Symfony va hydrater l'objet Groupe des données entrée dans le formulaire.

Les contraintes de validation

Dans Symfony, c'est Doctrine qui s'occupe de la validation. En effet, ces contraintes qui permettent de valider ou non les entrées utilisateur sont déclarées directement dans les entités.

De fait, elles sont donc très simples à mettre en place. Voici un exemple avec un attribut correspondant à un lien d'image, qui est donc de type String, mais qui, au moment de l'envoi du formulaire, sera traité comme un fichier.

```
{
    /**
     * @var string
     *
     * @ORM\Column(name="photoExemplaire", type="string", length=150,
     *                 nullable=false)
     *
     * @Assert\Image(
     *      minWidth = 400,
     *      maxWidth = 800,
     *      minHeight = 400,
     *      maxHeight = 600,
     *      allowPortrait = false
     * )
     *
     */
    private $photoexemplaire;
}
```

Cela ne parait pas, mais ici, tout est validation :

- Cet attribut ne peut pas être nul (`nullable=false`). Il doit donc obligatoirement être renseigné par l'utilisateur.
- Il a une longueur maximale de 150 caractères. Comme c'est nous qui générons un nom unique pour ce fichier avant de le bouger dans le bon répertoire cela n'a que peu d'importance.
- Ce fichier doit être une image. Il doit donc avoir le type MIME d'une image.
- Une largeur minimale de 400 pixels, et une maximale de 800 pixels.
- Une hauteur minimale de 400 pixels, et une maximale de 600 pixels.
- Enfin, il ne doit pas être en mode Portrait.

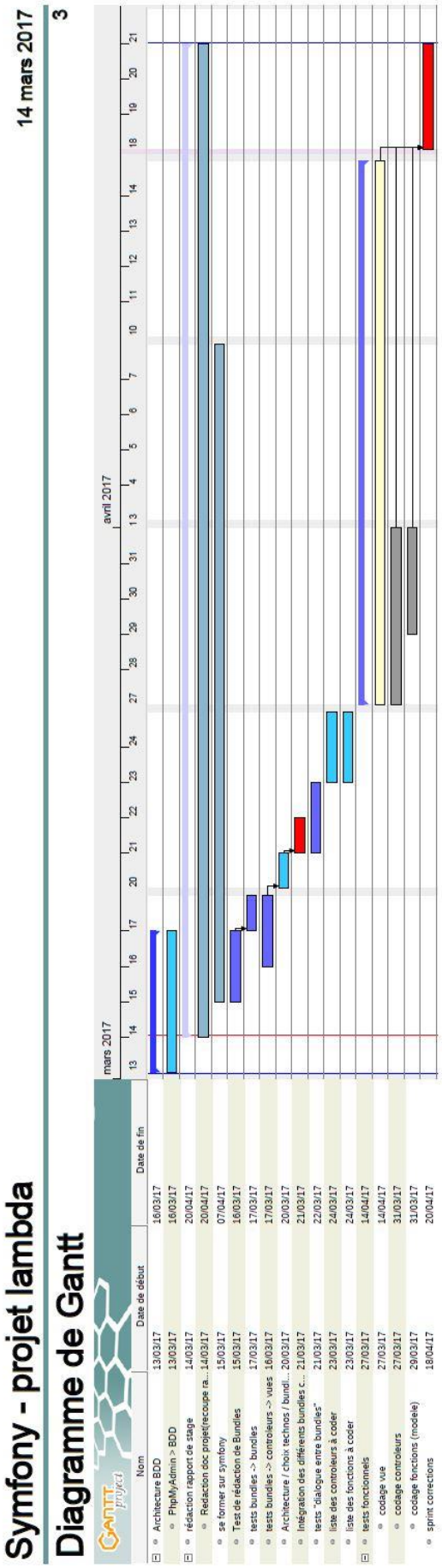
Si l'une de ces conditions n'est pas respectée, le formulaire ne sera pas valide. Bien sûr il y a des contraintes plus simples, comme les contraintes d'unicité ; et d'autres plus complexes, comme les expressions régulières destinées à reconnaitre un email, ou les choix quand à la longueur et la complexité des mots de passe.

Il existe beaucoup de contraintes disponibles, même une qui détecte si le fichier est corrompu, dans le cas d'un upload. Je vous invite à lire la documentation si vous souhaitez toutes les connaître.

Dans tous les cas, la validation se fait directement dans les entités. Ce qui est aussi un gain de temps, et des contrôleurs plus épurés.

Annexes

Diagramme de Gantt



Schémas de Base de Données

Schéma de tables (relationnel) :

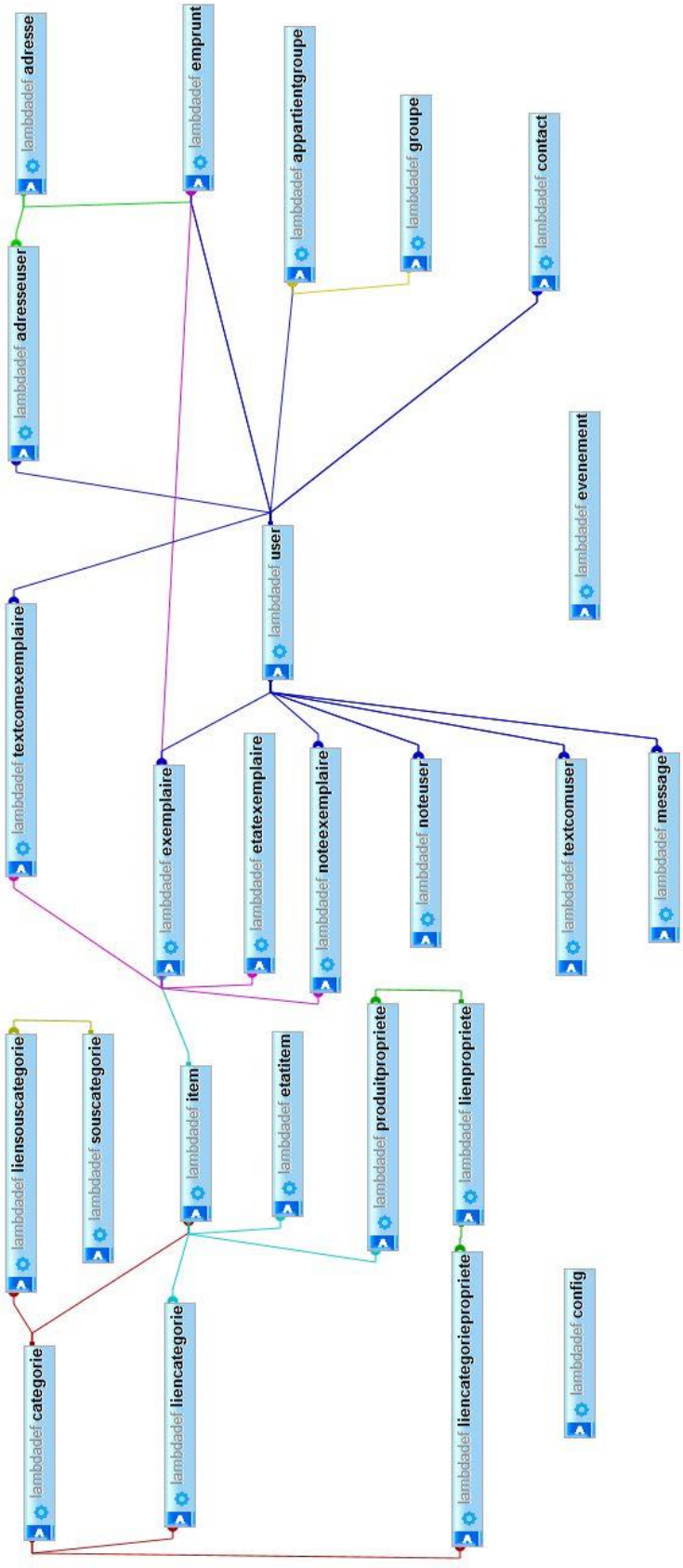


Schéma complet :