



Alphorm.com

Symfony 3, les fondamentaux

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Présentation du formateur
- Pourquoi Symfony ?
- Symfony 2 ou 3 ?
- Les objectifs du cours
- Le plan du cours
- Une application pour jouer avec l'environnement Symfony
- A qui s'adresse la formation ?
- Les pré-requis



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Présentation

- Michel CADENNES

- Développeur et architecte d'information indépendant
- Orientation : gestion des connaissances, intelligence artificielle, web sémantique



- Mes profils :

- LinkedIn : <https://www.linkedin.com/in/michel-cadennes-2a287726>
- Hopwork : <https://www.hopwork.fr/profile/michelcadennes>
- Twitter : <https://www.twitter.com/tchevengour>
- Github : <https://github.com/Septentrion>

Pourquoi Symfony ?

- PHP est la « *lingua franca* » du web
- Symfony 2 est le **premier framework** « moderne » pour PHP
- L'environnement offre une **vision architecturale claire**
- C'est **une boîte à outils** et non un CMS (voire un CMF)
- Les composants sont **de plus en plus utilisés** dans nombre d'outils PHP : Drupal, Laravel, Goutte, PHPBB, Piwik,...
- Symfony encourage **les bonnes pratiques** d'équipe de développement

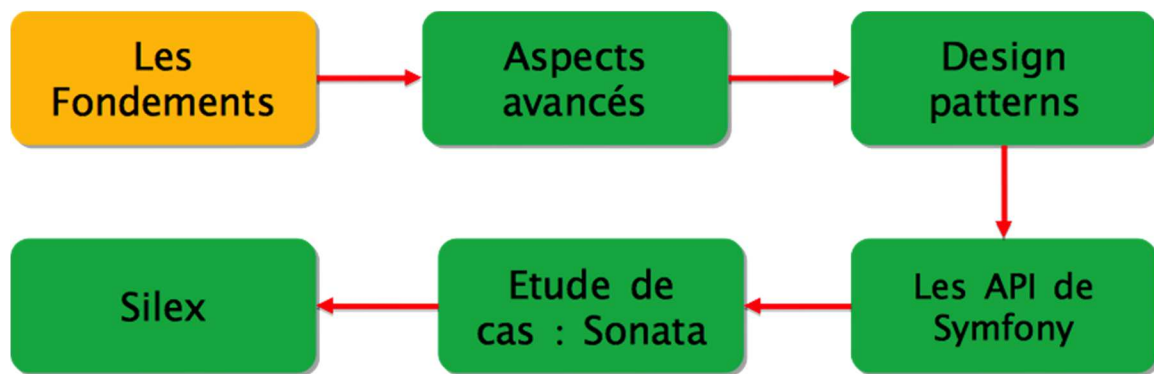
Symfony 2 ou 3 ?

- Les **versions supportées** actuellement sont la **2.7** et la **2.8**
- La philosophie de **SensioLabs** a été depuis la version 2.3 de ne pas retirer les fonctions déclarées obsolètes
- La version 3.0 procède principalement à un **nettoyage du code source**
- La très grande majorité du code **compatible avec les versions 2.x** récentes le sera encore avec la version 3
- La **version 3.0 n'est pas LTS** : des changements peuvent encore intervenir dans les mois qui viennent.

Objectifs

- Le but de ce cours est d'expliquer les fondements de **l'architecture d'une application** dans l'environnement Symfony :
 - Découvrir les **composants principaux** proposés par Symfony
 - Comprendre le **cycle d'exécution d'une requête**
 - **Créer et organiser son code** pour construire une application simple
 - Repérer les **bonnes pratiques** de Symfony et de PHP en général
 - Intégrer les **fonctionnalités avancées de PHP 5.3+** et les contraintes de Symfony

Coursus Symfony



Plan du cours

- Présentation
- PHP avancé
- Composer
- Prise en main
- Les bundles
- Le routeur
- Les contrôleurs
- Le moteur de template (Twig)
- Les formulaires
- Les modèles
- Les services
- Conclusion

Application à construire pendant la formation

- Utiliser Symfony pour développer une application-jouet : nous avons besoin de **gérer une médiathèque**.
- Ceci suppose :
 - Un modèle riche : documents de plusieurs types, utilisateurs, emprunts, événements,
 - Une interface publique + une administration
 - Des formats de présentation divers
 - Des scénarios d'utilisation variés
 - Des assistants nécessaires pour les différentes tâches
 - Des vérifications sur les informations transmises
 - Une politique d'URL pour les moteurs de recherche

A qui s'adresse la formation ?

- Aux développeurs
- Aux architectes
- Aux chefs de projets (techniques)

Pré-requis

- Une pratique de **PHP**
- Une connaissance des bases de la **Programmation Orientée Objet**
- Savoir configurer un serveur web, ou un environnement virtuel de type **XAMPP** (Linux) ou **WAMP** (Windows)



C'est parti !



Alphorm.com

Rappels sur PHP Fonctionnalités avancées

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Les différents types de classes en PHP5
- Les méthodes magiques
- Les espaces de noms



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Les classes abstraites

- Les classes définies comme **abstraites** ne peuvent pas être instanciées, et toute classe contenant au moins une méthode abstraite doit elle-même aussi être abstraite.
- Les méthodes définies comme abstraites déclarent simplement la signature de la méthode - elles ne peuvent définir son implémentation

```
abstract class T {  
    abstract function f()  
}
```

Les interfaces

- Les interfaces objet vous permettent de créer du code qui spécifie quelles méthodes une classe doit implémenter, sans avoir à définir comment ces méthodes fonctionneront.

```
interface T { ... }  
  
class A implements T {  
}
```

Les traits

- Les traits sont un mécanisme de **réutilisation de code** dans un langage à héritage simple tel que PHP.
- Un trait tente de réduire certaines limites de l'héritage simple, en autorisant le développeur à réutiliser un certain nombre de méthodes dans des classes indépendantes. La sémantique entre les classes et les traits, réduit la complexité et évite les problèmes typiques de l'héritage multiple et des Mixins.

```
trait T { ... }  
  
class A{  
    use T;  
}
```

Les classes anonymes

- Le support pour les classes anonymes a été rajouté en **PHP 7**.
- Les classes anonymes sont utiles lorsque de simples et uniques objets ont besoin d'être créés.

```
// PHP 7+ code  
$util->setLogger(new class {  
    public function log($msg)  
    {  
        echo $msg;  
    }  
});
```

Les méthodes magiques

- Les « méthodes magiques », qui sont toujours préfixées par '__' sont utilisées par PHP pour exécuter automatiquement du code lorsqu'un événement se produit.

`__construct()`, `__destruct()`, `__call()`, `__callStatic()`, `__get()`, `__set()`,
`__isset()`, `__unset()`, `__sleep()`, `__wakeup()`, `__toString()`,
`__invoke()`, `__set_state()` `__clone()` et `__debugInfo()`

Les espaces de noms

- Dans le monde PHP, les espaces de noms sont conçus pour résoudre deux problèmes que rencontrent les auteurs de bibliothèques et d'applications lors de la réutilisation d'éléments tels que des classes ou des bibliothèques de fonctions :
 1. **Collisions de noms** entre le code que vous créez, les classes, fonctions ou constantes internes de PHP, ou celles de bibliothèques tierces.
 2. La capacité de faire des **alias** ou de **raccourcir** des Noms_Extremement_Long pour aider à la résolution du premier problème, et améliorer la lisibilité du code.

Les espaces de noms

- Les espaces de noms sont déclarés avec le mot-clef :

```
namespace App\Dossier\SousDossier
```

- Les segments de l'espace de nom sont séparés par un '\'
- La déclaration de l'espace de nom est toujours sur la première ligne du fichier PHP (exception faite de la directive declare)
- On déclare un espace de noms par fichier source
- Le libellé de l'espace de nom respecte l'arborescence des dossiers

Les espaces de noms

- Une classe est appelée avec son espace de noms :

```
use App\Dossier\SousDossier\Classe
```

- et peut être liée à un alias qui sera ensuite utilisé dans le code :

```
use App\Dossier\SousDossier\Classe as C1
```

- L'espace de noms peut être retrouvé via la constante :

```
__NAMESPACE__
```

L'autoload

- Les espaces de noms ne sont pas (eux) magiques.
- Leur rôle est de discriminer les noms.
- Pour trouver et charger la classe, il faut, en plus, écrire une fonction `__autoload()` qui, en fonction de l'espace de nom, saura retrouver le fichier à un placement donné sur le serveur.

Ce qu'on a vu

- Le système des classes en PHP ≥ 5.3
- Les ajouts (plus ou moins) récents permettant de faire de la programmation fonctionnelle en PHP
- Le mécanisme des espaces de noms et pourquoi Symfony 2/3 ne fonctionne qu'avec PHP $\geq 5.3.9$





Alphorm.com

Rappels sur PHP Vers un style fonctionnel avec PHP 5

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Les fonctions anonymes
- Les itérateurs
 - L'interface **Iterator**
- Les générateurs
 - Les variables **static**
- Fonctions d'ordre supérieur



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Les fonctions anonymes

- Les fonctions anonymes, aussi appelées **fermetures** ou **closures** permettent la création de fonctions sans préciser leur nom.
- Elles sont particulièrement utiles comme fonctions de rappel, mais leur utilisation n'est pas limitée à ce seul usage.

```
$carre = function ($x) {  
    return x * x;  
};  
  
echo $carre(4);
```

Les fonctions anonymes

- Une fonction anonyme peut capturer des variables du contexte parent

```
$message = "world";  
$example = function ($arg) use ($message) {  
    var_dump($arg . ' ' . $message);  
};  
$example("hello");
```

Les fonctions anonymes

- Une fonction peut également retourner une fonction anonyme :

```
function setData($user, $pass, $host){  
    return function() use ($user, $pass, $host){  
        return array($user, $pass, $host);  
    };  
}  
  
$container = setData('test', 'password', 'localhost');  
var_dump($container());
```

Les itérateurs

- PHP 5 admet que des objets implémentent l'interface **Iterator**, et soient donc « *traversables* », c'est-à-dire parcourus par une boucle.

```
Iterator extends Traversable {  
  
    /* Méthodes */  
    abstract public mixed current ( void )  
    abstract public scalar key ( void )  
    abstract public void next ( void )  
    abstract public void rewind ( void )  
    abstract public boolean valid ( void )  
}
```

Les itérateurs

- PHP 5 vous laisse la responsabilité d'implémenter les différentes méthodes.
- En revanche, il existe un bouquet d'itérateurs particuliers de la bibliothèque standard pour répondre à des besoins spécifiques :

```
ArrayIterator,  
FileSystemIterator,  
RegexIterator,  
InfiniteIterator,  
  
etc.
```

Les générateurs

- Si les itérateurs sont des objets complexes, les générateurs sont des fonctions que l'on peut assimiler à des « **streams** », particulièrement adaptés lorsque l'on doit parcourir des grands ensembles séquentiellement.
- Un générateur utilise le mot-clef **yield** à la place de **return**

```
function pairs($x){  
    while true {  
        $x += 2;  
        yield $x;  
    }  
  
$y = pairs(0); // $y est une fonction
```

Les générateurs

- Il est possible d'envoyer des valeurs aux générateurs, pour créer des coroutines.

```
function powers($exposant) {  
    $x = 0;  
    while (true) {  
        $command = (yield $x);  
        if ($command == 'end') return;  
    }  
}  
  
$printer = powers(2); // affiche les carrés  
$printer->next();  
$printer->send('end');
```

Les variables « static »

- Une variable déclarée **static** dans le corps d'une fonction n'est évaluée que lors du premier appel de la fonction, a une portée locale, mais fait en sorte que le contexte d'exécution persiste à la fin de l'exécution de la fonction.

```
function test()  
{  
    static $a = 0;  
    echo $a;  
    $a++;  
}
```

Les variables « static »

- Les variables statiques servent notamment lors de la définition de fonctions récursives, mais sont aussi un moyen commode d'écrire des générateurs.

```
function fact($x)
{
    static $resultat = 1;

    if ($x == 0 ) {
        return $resultat;
    } else {
        return $x * fact($x - 1);
    }
}
```

Les fonctions d'ordre supérieur

- Assez souvent négligées, il est possible d'utiliser les fonctions **map**, **filter** et **reduce** sur des tableaux.
 - **map** applique une fonction à tous les éléments du tableau
 - **filter** retourne les éléments du tableau qui vérifient une certaine fonction
 - **reduce** calcule une valeur en appliquant itérativement une fonction à tous les éléments du tableau.

N.B. : Ces fonctions sont composables, à l'exception de reduce, qui doit toujours être finale.

Les fonctions d'ordre supérieur

- Sur les itérables en général :
 - **map** s'applique avec la fonction **iterator_apply**
 - **filter** peut être appliqué au travers de la classe **FilterIterator**
 - **reduce** n'a pas d'équivalent.

N.B. : On atteint les limites de PHP 5 en tant que
langage de programmation moderne.

Ce qu'on a vu

- Les fonctions anonymes
- Les itérateurs
- Les générateurs
- Les fonctions d'ordre supérieur





Alphorm.com

Rappels sur PHP PHP et PSR-x

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- L'initiative PSR
 - Les différents niveaux de normalisation



Formation Symfony3, les fondamentaux

alphorm.com™ ©

PSR

- PSR est une initiative d'un consortium d'acteurs de l'univers PHP (*PHP Framework Interop Group PHP-FIG*) pour formaliser des conventions d'écriture permettant la réutilisabilité du code entre les plate-formes.
- A ce jour, il y a **7 recommandations** et d'autres sont en cours de rédaction

PSR-4

- PSR-4 (ex PSR-0) est sans doute la recommandation la plus importante puisqu'elle précise comment se fait l'**autoloading** des classes dans une application PHP

PSR-1

- PSR-1 liste **des conventions d'écriture de base**, notamment :
 - Les fichiers PHP doivent s'abstenir de tout effet de bord, en particulier par le biais de require, include, echo, global, etc.
 - Les noms de classes s'écrivent en « capitales attractives », i.e. l'initiale est une majuscule
 - Les noms de méthodes s'écrivent en « camelCase » i.e. l'initiale est une minuscule

PSR-2

- PSR-2 précise **les conventions d'écriture du code**. Elle a pour vocation de rendre le code PHP plus lisible. Notamment :
 - Les indentations utilisent des espaces et non des tabs
 - Toutes les propriétés et les méthodes doivent avoir une visibilité déclarée
 - La parenthèse ouvrante du corps des méthodes est placée sur la ligne suivante

PSR-3

- PSR-3 contient un certain nombre de règles à propos des journaux de logs des applications. En particulier, elle définit les niveaux d'alerte et ce que devraient être les messages et la prise en compte du contexte
- PSR-3 fournit une classe abstraite : **AbstractLogger**

PSR-7

- PSR-7 liste des recommandations relatives aux messages HTTP.
 - Spécification d'interfaces pour les requêtes et les réponses
 - Les noms d'entêtes HTTP sont insensibles à la casse
 - Il est fourni une méthode pour gérer les entêtes à valeurs multiples
 - L'entête Host doit être fourni
 - Il devrait être fourni une interface pour des streams en cas de contenu très volumineux

Drafts

- D'autres recommandations sont en cours d'écriture. Notamment :
 - PSR-5 sur la documentation
 - PSR-11 sur l'injection de dépendances
 - PSR-12, qui devrait synthétiser PSR-1 & 2

Conclusion

- Dans les applications web, l'architecture « réelle » est souvent plus proche d'une architecture 4-tier (4 étages) que d'un MVC standard
- Cette architecture n'est pas hégémonique : Drupal n'est pas conçu ainsi et se rapprocherait plutôt d'une architecture 3-tier
- Ces modèles sont sans doute amenés à évoluer avec la tendance aux « clients lourds » de bibliothèques JavaScript (Angular, React) capables de prendre en charge une grande partie du traitement, et l'arrivée des WebComponents

Ce qu'on a vu

- L'initiative PSR
 - Les différents niveaux de normalisation



Alphorm.com

Rappels sur PHP
PHP et MVC

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- L'architecture MVC
 - Adéquation aux applications web



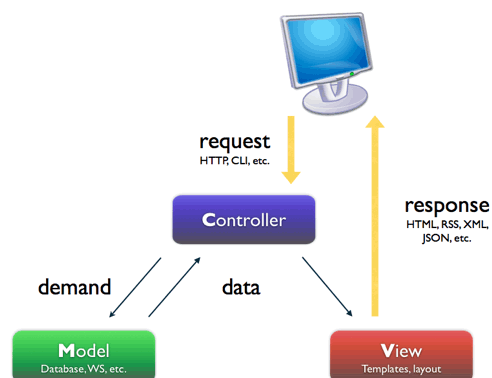
Architecture

- Une architecture pour les applications web

M = Modèle

V = Vue

C = Contrôleur



Le contrôleur

- Le contrôleur est chargé de gérer les interactions entre l'utilisateur et le modèle.
- Il reçoit les requêtes
- Il traduit la requête, soit en la transmettant au modèle, soit en demandant à la vue une mise à jour de l'affichage
- Il peut implémenter des fonctions de mise à jour de la vue

Le modèle

- Le modèle contient toutes les données relatives à l'application et toutes les méthodes pour y accéder.
- Il enregistre les vues et les contrôleurs qui doivent accéder aux données
- Il notifie les composants des modifications

La vue

- La vue est responsable de la présentation des données du modèle dans l'interface utilisateur
- Elle crée et initialise les contrôleurs
- Elle consulte les données du modèle
- Elle enregistre les actions de l'utilisateur et les transmet au contrôleur

Avantages

- L'architecture MVC schématise des rapports très clairs entre les différents composants de l'application
- Son fonctionnement n'est pas hiérarchique, ce qui autorise les vues à envoyer des requêtes directement au modèle (cf. **Doctrine** et **Twig**)
- Elle facilite le partage des tâches en matière de développement (entre les différents métiers)
- Les relations entre les composants sont *à priori* asynchrones.

Limites

- MVC est surtout conçu pour gérer le rafraîchissement simultané de plusieurs vues, ce qui n'est pas le cas des applications web (peut-être à venir avec les WebSockets)
- Dans les applications web, il y a une ambiguïté sur les vues qui sont à la fois des pages et le code exécuté par le navigateur **et** le code côté serveur chargé de composer l'interface (les moteurs de template)
- Une approche moderne de contrôleurs minces laisse de côté toute une partie du code qui ne se retrouve pas non plus dans le modèle (i.e. les services dans Symfony)
- L'interprétation de MVC varie énormément entre les implémentations

Conclusion

- Dans les applications web, l'architecture « réelle » est souvent plus proche d'une architecture 4-tier (4 étages) que d'un MVC standard
- Cette architecture n'est pas hégémonique : Drupal n'est pas conçu ainsi et se rapprocherait plutôt d'une architecture 3-tier
- Ces modèles sont sans doute amenés à évoluer avec la tendance aux « clients lourds » de bibliothèques JavaScript (Angular, React) capables de prendre en charge une grande partie du traitement, et l'arrivée des WebComponents



Alphorm.com

Composer Gestion de dépendances pour PHP

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Qu'est-ce que Composer ?
- Installer Composer
- Configuration de projet dans composer.json
 - La section « require »
 - Les contraintes sur les versions
 - Gestion de l'autoloader



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Composer

Introduction à la gestion de paquets

Qu'est-ce que Composer ?

- Composer est le gestionnaire de paquets et de dépendances pour PHP
- Est inspiré de npm (NodeJS) et bundler (Ruby)
- Est destiné à gérer l'installation de ressources par projet (et non globalement comme get-apt, pip ou gem)
- Vérifie les dépendances récursivement

Installer Composer

- Installer Composer se fait par la ligne de commande (Linux | OS X) :

```
curl -sS https://getcomposer.org/installer | php
```

- Si vous voulez rendre Composer exécutable globalement, il faut ensuite le déplacer dans un répertoire inclus dans \$PATH

```
mv composer.phar /usr/local/bin/composer
```

Utiliser Composer

- Une fois Composer installé, il suffit de l'appeler depuis la ligne de commande :

```
composer --version
```

- L'initialisation d'un projet PHP se fait par la commande :

```
composer install
```

Configurer un projet

Le schéma de composer.json

composer.json

- Pour initialiser un projet, Composer se sert d'un fichier de configuration nommé composer.json
- Ce fichier décrit toutes les dépendances sur lesquelles repose votre projet, ainsi que diverses méta-données.
- La section principale trouvée dans ce fichier est : require, qui indique quelles sont les ressources à installer

```
{  
    "require": {  
        "monolog/monolog": "1.0.*"  
    }  
}
```

composer.json : require

- Les paquets requis par require ont un nom et une version.
- Par convention le nom comporte deux segments, le premier étant le nom du « vendor » et le second le nom du projet

`monolog/monolog`

composer.json : require

- Pour les versions, il est possible d'utiliser une expression logique à base de la syntaxe suivante :
 - un numéro de version sur trois segments : 1.3.25
 - des opérateurs logiques : < > <= >= , || |=
 - des tirets (pour indiquer un intervalle)
 - des astérisques (comme joker)
 - des tilde (pour indiquer l'incrément de correctif le plus récent)
 - des circonflexes (pour indiquer la version mineur la plus récente)
 - un suffixe -dev ou -stable pour désigner les statut de la version

composer.json : require

- Exemples :
 - 1.1.*
 - $\geq 1.2, < 1.3$ (équivalent à $\wedge 1.2$)
 - 1.0 - 2.0 (équivalent à $\geq 1.0.0 < 2.1$)
 - ~ 1.2 (équivalent à $\geq 1.2 < 2.0.0$)
 - $\geq 2.7, \leq 2.8 \parallel \geq 3.1$

composer.json : require

- Il est aussi possible de préciser une branche (sur Github par exemple), en préfixant le nom de la branche par 'dev-'
 - dev-master
- Il est aussi possible également de donner des contraintes de stabilité (par défaut stable), la convention étant :
 - dev > alpha > beta > RC > stable
 - 1.2@beta

L'autoload

- Composer crée un fichier d'autoload inclus dans le dossier vendor.
- Vous pouvez utiliser directement les classes en intégrant dans votre programme PHP :

```
require __DIR__ . '/vendor/autoload.php';
```

L'autoload

- Il est même possible de déclarer son propre autoloader, par le biais du label autoload.

```
{  
    "autoload": {  
        "psr-4": {"Acme\\": "src/"}  
    }  
}
```

Ce qu'on a vu

- Le rôle de Composer dans les projets PHP
- Installer Composer
- Utiliser Composer via le shell
- Créer un fichier de configuration simple



Alphorm.com

Composer
Options et commandes
de Composer

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Les options du fichier composer.json
- Les commandes principales
- Les options de configuration
- Les dépôts de paquets



Les options de Composer.json

Un fichier de configuration enrichi

Options de composer.json

- Le schéma de composer.json comporte encore d'autres sections :
 - pour identifier l'application
 - pour spécifier le niveau de stabilité d'un paquet
 - pour identifier les dépôts sources
 - pour identifier des scripts à exécuter après Composer
 - pour passer des informations annexes
 - pour ajouter des paquets suggérés ou requis en phase développement

Options de composer.json

- Certaines options ne peuvent se trouver qu'à la racine du fichier de configuration dont :

scripts, config, minimum-stability, require-dev

Les commandes

Piloter Composer depuis la ligne de commande

Les commandes de Composer

- Outre install, Composer possède une liste de commandes pour divers usages.
- Par exemple :
 - init pour créer de manière interactive le fichier composer.json
 - update pour mettre à jour les paquets
 - remove pour désinstaller un paquet
 - search pour chercher un paquet dans un dépôt
 - suggests pour lister les dépendances suggérées par les paquets
 - validate pour valider le fichier composer.json
 - self-update pour mettre à jour Composer

Les options de configuration

- Le schéma de composer.json comporte une section « config » qui permet de paramétrer Composer.
- Entre autres :
 - use-include-path autoload inspectant de PATH de PHP
 - preferred-install source ou dist
 - platform choisir une version de PHP
 - search pour chercher un paquet dans un dépôt
 - vendor-dir répertoire pour les paquets tiers
 - notify-on-install notifier le dépôt de l'installation du paquet

Les dépôts

Les dépôts

Composer repose principalement sur le dépôt Packagist

<http://www.packagist.org>

On peut aussi trouver des ressources ici :

<http://www.phppackages.org>

Et ici pour Symfony en particulier :

<http://knpbundles.com/>

Les dépôts

- Paramétrer les sources d'où tirer les paquets se fait dans la section 'repositories'.
- En particulier, il est possible d'utiliser Github ou PEAR comme sources.

```
{
  "repositories": [
    {
      "type": "vcs",
      "url": "https://github.com/igorw/monolog"
    }
  ],
  "require": {
    "monolog/monolog": "dev-bugfix"
  }
}
```

Les dépôts : PEAR

- Un exemple PEAR, les paquets sont préfixés par 'pear-<channel>' :

```
{
  "repositories": [
    {
      "type": "pear",
      "url": "https://pear2.php.net"
    }
  ],
  "require": {
    "pear-pear2.php.net/PEAR2_Text_Markdown": "*",
    "pear-pear2/PEAR2_HTTP_Request": "*"
  }
}
```

Les dépôts : composer

- Composer peut aussi importer des ressources indirectement, en allant chercher un autre fichier de déclaration packages.json qui doit être présent sur le dépôt.

```
{
  "repositories": [
    {
      "type": "composer",
      "url": "https://www.<exemple>.com"
    }
  ],
}
```

Ce qu'on a vu

- Configurer Composer
- Les diverses commandes pour maintenir un projet avec Composer
- Où trouver des dépôts



Pour aller plus loin

- La documentation de Composer

<https://getcomposer.org/doc/>



Symfony Installation de Symfony 3

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Installer Symfony
 - Tester la configuration serveur
 - Lancer le serveur intégré
 - La première page
- Les outils du développeur
 - La barre d'outils web
 - Le profileur



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Les pré-requis pour faire fonctionner Symfony 3

- PHP 5.5.9+
- Support de JSON
- Support de ctype
- **php.ini** doit avoir une option **date.timezone** définie
(une erreur extrêmement courante)

Installer Symfony

- Trois techniques pour installer Symfony :
 - Télécharger l'archive depuis **Github**
 - Utiliser l'**installeur** symfony
 - Utiliser **Composer**

Installer Symfony : Téléchargement

- La méthode la plus directe
- Installe tout sans distinction

Installer Symfony : symfony

- Nécessite d'installer un script (une fois pour toute)
- Permet de systématiser les installations
- Offre davantage de souplesse dans la création de projets
- La méthode **recommandée par SensioLabs**

Installer Symfony : Composer

- Nécessite d'installer **Composer**
- Permet d'installer sélectivement des composants de Symfony
- Permet d'installer simultanément des bibliothèques tierces
- Offre un outil complet pour la **gestion des projets**

Lancer Symfony

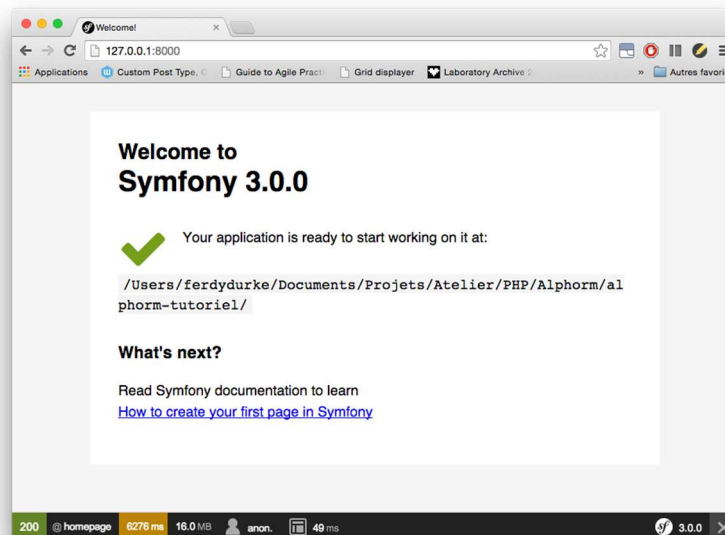
- Une fois tous les fichiers installés, il est possible de démarrer Symfony de manière autonome en utilisant le serveur HTTP intégré (*celui de PHP, en fait*)

```
php bin/console server:start
```

- Pour l'arrêter :

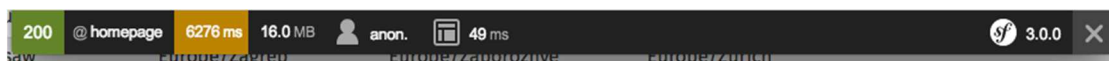
```
php bin/console server:stop
```

La première page d'un projet Symfony



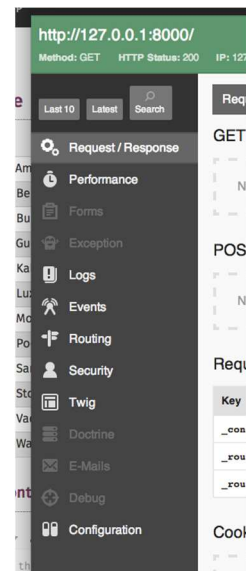
La barre d'outils web

- La barre d'outils de Symfony s'affiche pour donner des informations sur le déroulement de la requête.



Le profileur

- Par la barre d'outils, le développeur a accès à des données de profilage très détaillées



Ce qu'on a vu

- Installer le code de Symfony
- Tester le serveur
- Démarrer simplement Symfony
- Explorer les données fournies par le profileur





Alphorm.com

Symfony Architecture d'un projet Symfony

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Architecture d'un projet Symfony
- Symfony comme boîte à outils
- Les fichiers de configuration
- Les formats de fichiers dans Symfony
- Introduction aux environnements

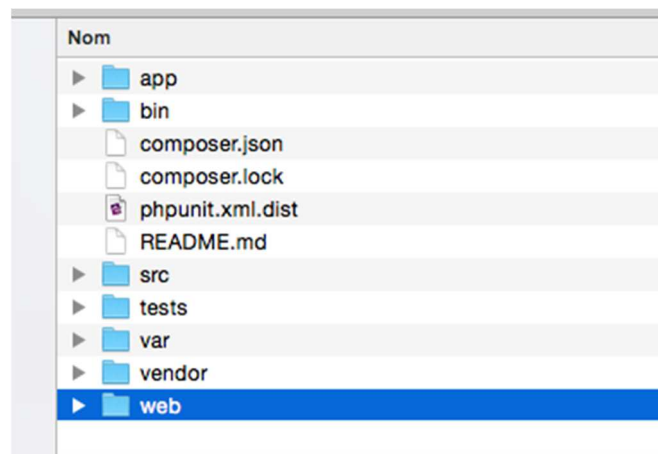


Formation Symfony3, les fondamentaux

alphorm.com™ ©

Tour d'horizon de l'installation

- Les répertoires à la racine de votre projet Symfony :



Symfony comme boîte à outils

- Symfony repose sur une collection de composants (**Components**), qui assurent les différentes fonctions du noyau du framework
 - Assetic
 - ClassLoader
 - Config
 - Console
 - CssSelector
 - Debug
 - DependencyInjection
 - DomCrawler
 - EventDispatcher
 - ExpressionLanguage
 - FileSystem
 - Finder
 - Form
 - HttpFoundation
 - HttpKernel
 - Intl
 - OptionsResolver
 - Process
 - PropertyAccess
 - Routing
 - Security
 - Serializer
 - Stopwatch
 - Templating
 - Translation
 - VarDumper
 - Yaml

Les fichiers de configuration

- Le répertoire `app/config` contient toutes les données globales de configuration du projet Symfony

Les formats de configuration

- D'une manière générale, Symfony laisse beaucoup de choix au développeur. Les quatre formats *a priori* reconnus sont :
 - **YAML** : un format simple, léger et aisément compréhensible par des non-développeurs
 - **XML** : un format robuste, reconnu par PHP et adapté aux manipulations par les machines (XSLT, Dom, etc.)
 - **Annotations** : A l'instar de PHPDoc, permettent d'intégrer les configurations directement dans les fichiers sources
 - **PHP** : le langage natif

Les environnements

- Un environnement est globalement une configuration de Symfony.
- Trois environnements sont définis par défaut :
 - **prod** : ce que voient les utilisateurs du site
 - **dev** : ce que voient les développeurs du site
 - **test** : ce qui est utilisé pour les jeux de tests

Ce qu'on a vu

- L'organisation du système de fichiers de Symfony
- Les composants qui constituent la boîte à outils de Symfony
- La configuration globale de l'application
- Les types de formats de fichiers supportés par Symfony
- Comment tirer parti des environnements





Alphorm.com

Symfony

De Symfony 2 à Symfony 3

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Quelques changements
- Une règle : ne pas casser la rétro-compatibilité
- Migrer d'une version 2.x à 3.0
- Quelle version utiliser ?

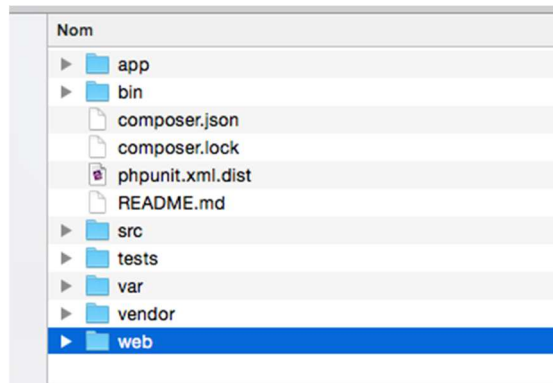


Formation Symfony3, les fondamentaux

alphorm.com™ ©

Tour d'horizon de l'installation

- De nouveaux répertoires :
 - /var
 - /tests



Liste des modifications

- Les listes des modifications apportées au code de Symfony est disponible dans le fichier :

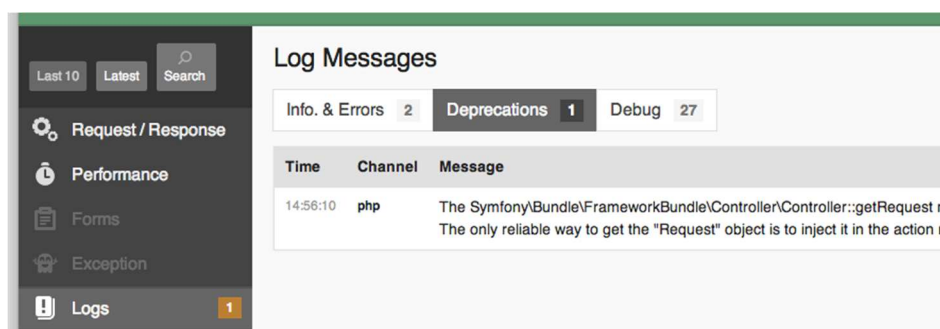
`/vendor/symfony/symfony/UPGRADE-3.0.md`

La rétro-compatibilité

- La version 3.0 enlève toutes les fonctions devenues obsolètes.
- Celles-ci vont donc provoquer des erreurs si elles restent dans le code

Utiliser le profileur

- Le profileur affiche des warnings pour toutes les fonctions **obsolètes**, depuis la version 2.5



Migrer de la version 2 à la version 3

- Installer une **version 2.8**
- Rechercher le code obsolète au travers du profileur
- Eventuellement rechercher le code obsolète avec PHPUnit (et le bridge développé par Symfony)
- Modifier le code pour le rendre *up-to-date*
- Basculer toute la base de code propre vers l'installation **3.0**

LTS

- La version 3.0 ne sera pas une version à « support à long terme » (Long Time Support — LTS)
- La première version LTS pour Symfony 2 était le 2.3 (obsolète mi-2016), puis la 2.7 et 2.8
- La première version LTS pour Symfony 3 sera la 3.4 prévue mi-2017
- La version 3.0 et la version 2.8 sont **presque simultanées**
- La version 3.0 n'ajoute que très peu de fonctions qui ne seraient pas compatibles avec la version 2.8

Ce qu'on a vu

- Les changements « structurels » dans le code de Symfony 3
- Comment trouver les fonctions obsolètes
- Comment s'assurer que le code de l'application est compatible avec Symfony 3
- Les raisons d'utiliser une version plutôt qu'une autre



Alphorm.com

Les bundles

Créer des modules fonctionnels pour Symfony

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES

Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Qu'est-ce qu'un « **bundle** » ?
 - Les bundles tiers
 - Les bundles de l'application
- Création d'un bundle
- Architecture d'un bundle
- Installer un bundle tiers
 - Où trouver des ressources ?
- Hériter d'un bundle parent
- Bonnes pratiques pour les bundles



Qu'est-ce qu'un bundle ?

- Un « **bundle** », c'est littéralement un paquet, voire un fagot ou une botte.
- C'est donc l'idée d'un certain nombre de ressources qui sont liées ensemble pour créer **une unité fonctionnelle**.
- En l'occurrence, un bundle contient tout ce qu'il faut de l'architecture **MVC** (et plus) pour construire une application au-dessus du noyau de Symfony
- Les bundles sont des **unités architecturales et non techniques**. La décomposition de l'application en modules fonctionnels est donc un choix qui n'est pas unique

Les bundles tiers

- Les bundles tiers sont hébergés dans le répertoire : `/vendor`
- Il n'est pas conseillé de les modifier ou d'installer son propre code dans ce dossier, sauf si vous construisez des bibliothèques génériques.
- Les bundles tiers comportent souvent des moyens pour étendre leurs fonctionnalités (comme des *factories*, ou des *classes abstraites*)

Les bundles de l'application

- Les bundles spécifiques à l'application sont hébergés (par défaut) dans le répertoire : `/src`
- Il est possible de créer son propre dossier de bundles. Néanmoins, cela n'est pas une pratique courante et a un intérêt plutôt limité.

Créer un bundle

- Le moyen le plus simple de créer un bundle est de passer par la ligne de commande

```
php bin/console generate:bundle
```

- Symfony entre alors dans une procédure interactive qui vous permet de paramétrer votre bundle.
- Il est évidemment possible de créer manuellement un bundle ou de copier simplement un dossier existant dans le dossier [/src](#)

Créer un bundle

- La procédure via la console Symfony a l'avantage de modifier l'application pour que le bundle soit immédiatement actif.
- Cela veut dire, mettre à jour deux fichiers :

```
/app/appKernel.php  
  
/app/config/routing.yml
```

Les fondamentaux

- Une fois créé, le bundle contient au moins trois éléments essentiels :
 - Un dossier contenant les contrôleurs : `/Controller`
 - Un dossier contenant les routes : `/Resources/config/routing.____`
 - Une classe d'initialisation pour le bundle : `<nomDuBundle>Bundle.php`

Installer un bundle tiers

- Pour cela, la meilleur méthode est d'utiliser **Composer**.
- Puisque Symfony respecte les conventions PSR-x, on peut utiliser les packages qui sont sur ces sites :

<http://symfohub.com/bundles>

<http://www.packagist.com>

<http://www.knpbundles.com>

Choisir un bundle

- Il existe maintenant plusieurs milliers de bundles pour Symfony.
Quelques critères :
 - Les followers sur GitHub
 - Un bon README
 - Intègre Travis CI <https://scrutinizer-ci.com/>
 - Reconnu par Composer
 - Maintenu récemment
 - Nombre de dépendances

Bonnes pratiques

- **SensioLabs** recommande d'utiliser un seul bundle (**AppBundle**) pour le code métier — à discuter
- Il est possible de créer des espaces de noms en dehors des bundles pour du code métier (dans /src) — mais les bundles doivent rester privilégiés
- Les standards PSR-1 et PSR-2 devraient toujours être respectés
- Le bundle devait fournir des jeux de tests — pas abordés ici

Et maintenant ?

- Comme le dit Symfony, votre bundle est prêt à être utilisé.
- Reste une possibilité : déclarer un bundle parent dont vous pourrez hériter. Pour cela, il faut juste modifier le fichier :

```
<nomDuBundle>Bundle.php
```

Ce qu'on a vu

- Ce que c'est qu'un bundle et où trouver l'information
- Comment créer un bundle
- Comment installer des bundles tiers
- Les règles à respecter pour organiser un bundle
- Comment créer une hiérarchie de bundles





Alphorm.com

Le routage Introduction au routage

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

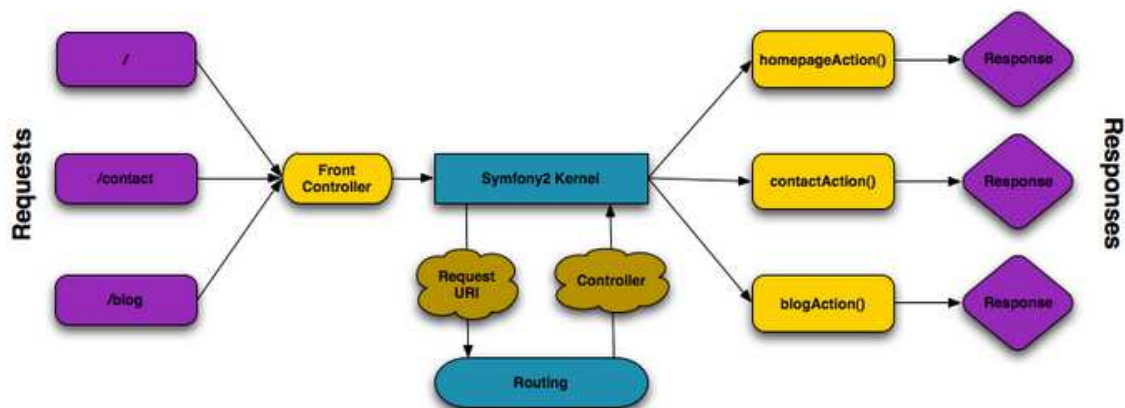
Plan

- Rappels sur le routage
- Intégration par défaut des routes dans Symfony
 - Personnaliser la configuration
- Différentes façons de déclarer les routes
 - Exemples
 - Avantages et inconvénients
- Relation entre route et contrôleur

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Le Routage dans Symfony



Les routes

- Les routes sont un élément essentiel des applications web.
 - Elles doivent être significatives pour l'utilisateur
 - Elles doivent être dénuées d'ambiguïté pour l'analyser de l'application
 - Elles doivent permettre une bonne indexation par les moteurs de recherche
 - Elles doivent être suffisamment simples pour être transmises
- Toute application web doit avoir une politique d'URL stricte et stable

Les URL

- Une URL se décompose en :
 - un protocole => **http**
 - un nom de domaine => **www.exemple.com**
 - un port => **8080**
 - un chemin => **/blog/article/** (les *slashes* découpent des segments)
 - un nom de ressource => **index.html, premier-billet, image.jpg**
 - des options => **?x=hello&y=25**
 - un fragment => **#chapitre-3**

Les URL perçues par Symfony

- Une URL se décompose en :
 - protocole => **fichier de routage**
 - nom de domaine => **Apache**
 - port => **Apache**
 - chemin => **fichier de routage**
 - nom de ressource => **fichier de routage**
 - options => contrôleur
 - fragment => navigateur

La réécriture d'URL

- Dans Symfony, le « **contrôleur principal** », ou point d'entrée de l'application est en fait multiple, en fonction de l'environnement.

```
app.php          // point d'entrée public  
app_dev.php      // point d'entrée de l'environnement de développement
```

- L'analyseur d'URL PHP de Symfony est maintenant considéré comme quasiment aussi efficace que l'analyseur d'Apache

Le composant Routing

- **Routing** est un des composants du noyau de Symfony. Il est en charge, en particulier de :
 - Gérer des collections de routes
 - Déterminer les contextes des requêtes
 - Apparier une URL et une route
 - Forger des URL valides en fonction d'une route

Les formats de fichiers de configuration

- Symfony laisse le développeur choisir son propre format de fichier pour les configurations. Il y a 4 possibilités :
 - **YAML** // Très simple et concis
 - **XML** // Très robuste
 - **Annotations** // Permettent d'embarquer tout dans un seul fichier
 - **PHP** // Ne nécessite aucun traitement syntaxique

YAML (Exemple)

```
blog:
  path:      /blog/{page}
  defaults: { _controller: AppBundle:Blog:index }
```

N.B. : Attention aux espaces !!

XML (Exemple)

```
<route id="blog" path="/blog/{page}">
  <default key="_controller">AppBundle:Blog:index</default>
</route>
```

N.B : Syntaxe un peu fastidieuse (néanmoins la solution la plus sûre,
employée tout au long de ce cours)

Annotations (Exemple)

```
/**
 * @Route("/blog/{page}")
 */
```

N.B. : Très pratique, mais nuit à la lisibilité des fichiers

PHP (Exemple)

```
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AppBundle:Blog:index',
)));
```

N.B. : Non déclaratif

Routeur et contrôleur

- Sur le web, les routes sont des API de votre application. Elles en sont le versant syntaxique. Le versant opérationnel, lié de manière indissoluble est le contrôleur.
- A chaque route doit correspondre une action de contrôleur :

```
public function operationAction (...)
```

- Toutes les méthodes représentant des actions liées à des routes doivent être suffixées par le mot-clef **Action**.

Ce qu'on a vu

- Dans cette vidéo, nous avons vu le principe du routage :
 - Le mécanisme repose sur un des composants de Symfony : **Routing**
 - Comment Symfony configure par défaut le mécanisme de routage
 - Les différentes syntaxes permettant de configurer les routes (et pas qu'elles)
 - Comment est assurée la liaison entre la structure de la route et le traitement de la requête



Dans les prochaines sessions, nous verrons toutes les possibilités offertes pour décrire les routes et les bonnes pratiques pour gérer ses configurations.



Alphorm.com

Le routage

Configuration des routes

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES

Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Création d'une route simple
 - Les éléments essentiels : nom & motif du chemin
 - L'élément <default />
- Ajouter des éléments variables dans une route
 - Éléments requis ou éléments optionnels
- Ajouter des contraintes aux variables
- Déclarer des méthodes HTTP
 - Déclarer un hôte HTTP
- Les variables spéciales

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Caractéristiques d'une route

- Une route est principalement définie par deux propriétés clefs :
 - **id** : un identifiant unique qui permet à l'application d'accéder à la route
 - ex : bundle_action
 - **path** : le schéma de l'URL qui permettra d'analyser, par pattern-matching, la requête du navigateur
 - ex : /blog/premier-billet

La propriété default

- Une autre propriété essentielle des routes est la propriété **default**, qui permet de préciser la valeur des paramètres donnés pour une route.
- En particulier, une de ces propriétés est le nom du **contrôleur** associé à la route.

```
_controller => <nomDuBundle>:<nomDuContrôleur>:<nomDeAction>
```

Des variables dans le schéma de route

- La propriété **path** contient des schémas d'URL.
- On souhaite donc que toutes celles répondant à un schéma soient capturées par la même règle.
- Pour ça, on introduit des variables dans le schéma :

```
path => /article/{mois}/{jour}/{titre}
```
- Les noms des variables doivent se retrouver comme arguments de la méthode du contrôleur

Des variables dans le schéma de route

- Par défaut les variables sont **requises**.
- On peut les rendre optionnelles en ajoutant une entrée dans la liste des **default**.

```
default => mois = 1
```

- Une autre technique (moins lisible) consiste à donner une valeur par défaut directement dans la signature de la méthode du contrôleur

Des contraintes sur les valeurs

- Il est également possible de poser des contraintes sur les valeurs des segments de la route, grâce à des **expressions régulières**. On se septuor cela de la propriété **requirement**.

requirement => mois : d{2} // Les mois sont sur deux chiffres

requirement => titre : [a-w\~]+ // les titres ssont composés de lettres et de tirets

Des contraintes sur les requêtes

- Une autre manière de distinguer les routes est de prendre en compte la **méthode HTTP** de la requête.
- On peut ainsi ajouter une propriété **methods** pour le spécifier :

```
methods => GET | POST
```

Des contraintes sur les requêtes

- Enfin, il est possible de filtrer les requêtes en fonction du domaine auquel on souhaite accéder.
- Cela est possible par le biais de la propriété **host**.

```
host => m.domaine.tld  
host => {sous_domaine}.domaine.tld // avec une variable
```

Des variables spéciales

- Hormis **_controller**, les routes reconnaissent deux variables spéciales (toujours préfixées par le caractère _)

_format => indique le format de la réponse : html, json, xml, etc.

_locale => indique la langue associée à la parce demandée

Ce qu'on a vu

- Dans cette session, nous nous sommes plus précisément attachés à la syntaxe des routes.
 - Comprendre le fonctionnement des motifs pour les routes
 - Ajouter des contraintes et des options pour aiguiller le routeur de Symfony
 - Ajouter des informations pour envoyer des **réponses** appropriées au navigateur

Pour aller plus loin

Dans la dernière session, ajouter quelques astuces pour gérer efficacement les routes et les implications sur une politique d'URL.



Alphorm.com

Le routage

Gestion des routes

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES

Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Importer des routes externes
- Règle d'application des routes
- Découper ses fichiers de routage
- Préfixer les routes d'un ensemble importé
- Reconfigurer la racine des routes
- Utiliser HTTPS
- Déboguer les routes via la ligne de commande

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Importer des routes

- Par défaut, les routes se trouvent dans les fichiers **routing.<fmt>** des bundles ou dans le fichier **routing.yml** de l'application.
- Il est toutefois recommandé de diviser les fichiers de configuration si ceux-ci deviennent trop volumineux. A ce moment, on peut utiliser la directive **import** pour indiquer des ressources externes.

Règle d'application

- Les schémas de routes s'appliquent selon la règle : « premier trouvé, premier servi ».
- De ce fait, il est très important :
 - de bien classer les routes
 - d'avoir une politique de schémas claire et lisible
 - d'utiliser les ressources syntaxiques pour aider les moteurs à discriminer correctement les URLs qu'ils reçoivent

Découper les fichiers

- Par défaut, les routes se trouvent dans les fichiers **routing.<fmt>** des bundles ou dans le fichier **routing.yml** de l'application.
- Il est toutefois recommandé de diviser les fichiers de configuration si ceux-ci deviennent trop volumineux. A ce moment, on peut utiliser la directive **import** pour indiquer des ressources externes.

Préfixer les routes

- Il est souvent de bonne pratique d'associer à tout un ensemble de routes un même segment.

/blog // Toutes les URL des billets de blog commencent par ce segment

- Il est possible d'indiquer cela si un ensemble de routes est déclaré comme ressource externe.

Reconfigurer la racine des routes

- Par défaut, la racine se trouve, comme nous l'avons vu dans les fichier **routing.yml** associé à l'application.
- Mais ceci peut être changé dans le fichier **config.yml**, soit pour changer l'emplacement de la racine, soit pour en changer le format, **par exemple**

Tester les routes

- Grâce à la ligne de commande, il est possible de tester que les routes configurées sont véritablement accessibles par le routeur.

```
php bin/console router:debug
```

Forcer la sécurité des requêtes

Parmi les options avancées, il est aussi possible de préciser le protocole utilisé :

```
schemes =>http | https
```

Ce qu'on a vu

- Dans cette session, nous avons exploré quelques considérations méthodologiques sur la manipulation des routes dans Symfony :
 - L'organisation des fichiers de routage
 - Comment éviter les conflits dans votre bibliothèque de routes
 - Une introduction aux problèmes de politique d'URL

Pour aller plus loin

- D'autres possibilités sont offertes par le composant **Routing**, qui sort de l'objet de ce cours, comme par exemple :
 - Configurer plus efficacement les routes grâce à des services
 - Créer des *loaders* ad hoc créer des routes de manière dynamique
 - Passer des paramètres supplémentaires au contrôleur



Alphorm.com

Les contrôleurs
Au cœur du cycle
d'exécution

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Introduction
 - Créer une classe de contrôleur
 - Où placer les classes de contrôleur ?
 - Précisions syntaxiques
- Créer une action en lien avec une route
- Le cycle d'exécution
 - La classe Request
 - La classe Response



Créer une classe de contrôleurs

- Une classe de contrôleur est une sous classe de la classe Controller :

```
class ExempleController extends Controller
```

- Implémenter la classe Controller permet d'avoir accès à toute une série de raccourcis syntaxiques, mais n'est pas obligatoire. Dans certains cas, on peut même utiliser la classe **ContainerAware** dont Controller hérite.

```
class ExempleController extends ContainerAware
```

Créer une classe de contrôleurs

- Le contrôleur a pour tâche spécifique de gérer la réponse à la requête. Celle-ci peut être généralement de deux types :
 - un **contenu envoyé** au navigateur ou
 - une **redirection**

Où placer les contrôleurs ?

- Les classes de contrôleurs sont placées dans le **dossier Controller** du bundle correspondant.
- A l'intérieur du dossier, il est possible de créer des sous-dossiers pour classer ses contrôleurs
- Lorsque l'on crée un bundle, une classe est immédiatement créée, au nom de **DefaultController**, mais son existence n'est pas du tout obligatoire.

Règles syntaxiques

- Les classes doivent porter le même nom, avec la même casse, que le fichier qui les contient
- Les classes sont toujours suffixées par **-Controller**
- Grâce aux espaces de noms, des classes dans des *bundles* différents peuvent porter le même nom

La signature du contrôleur

Un contrôleur est une méthode de la classe de contrôleurs

```
public function exempleAction()
```

Le suffixe **-Action** indique que cette méthode est liée à une route.

La signature reprend la liste des variables définies dans la route, sans ordre précis. des valeurs par défaut peuvent être attachées aux arguments.

```
public function exempleAction($id, $slug, $t = 0)
```

La classe Request

- La classe Request a pour principale fonction de fournir des accesseurs pour tous les paramètres gérés habituellement par PHP dans les variables super-globales.

```
public function indexAction(Request $request) {  
  
    var  
  
    $request->isXmlHttpRequest(); // Ajax ?  
    $request->getPreferredLanguage(array('en', 'fr'));  
    $request->query->get('page'); // $_GET  
    $request->request->get('page'); // $_POST  
}
```

La classe Response

- La classe **Response** a la responsabilité de gérer tous les paramètres de la réponse :

```
$reponse = new Response($contenu, $statutHTTP, $entetes);
```

- Les entêtes sont généralement associés séparément :

```
$reponse = new Response($contenu);  
$reponse->header->set('Content-Type', 'application/json');
```

Créer un contrôleur

- Le contrôleur le plus simple aurait une forme telle que celle-ci :

```
public function exempleAction(Request $request) {  
    return new Response('<h1>Hello ! </h1>');  
}
```

- Ne faisant que renvoyer du code HTML au navigateur. Ici tous les entêtes liés à la réponse sont pris en charge automatiquement.

Créer un contrôleur

```
public function exempleAction(Request $request, $id, $slug, $t = 0)
```

- Dans le cas où la requête est intégrée à la signature du contrôleur, il faut faire attention d'utiliser la découverte de type (*type hinting*) pour permettre à Symfony de repérer l'argument correspondant
- Tout contrôleur doit envoyer une réponse, sinon une erreur est déclenchée.

Ce qu'on a vu

- Comment construire une classe de contrôleurs
- Comment Symfony utilise les contrôleurs dans son cycle d'exécution
- Le fonctionnement de la classe Request
- Le fonctionnement de la classe Response
- Comment créer un contrôleur simple



Pour aller plus loin

- La prochaine étape consiste à écrire des contrôleurs pour le monde réel.
- Vous pouvez consulter la documentation de l'API Symfony pour avoir plus de détails sur les classes **Controller**, **Request** et **Response**

<http://api.symfony.com/3.0>



Alphorm.com

Les contrôleurs Les outils du contrôleur

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Récupérer les paramètres de la requête HTTP
- Options de la classe Response
- Autres possibilités : redirect, redirectToRoute, forward
- Gestion des erreurs et page 404
- Accéder à des « services »
 - Gestion de la session
 - Envoyer des messages Flash
- Bonnes pratiques pour les contrôleurs
 - Des actions maigres et des services



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Utiliser la classe Request

La classe Request permet d'explorer et de manipuler les requêtes HTTP, en offrant une interface avec les variables superglobales.

Exemples :

```
getHost()  
getContentType()  
isMethod()  
isMethodSafe()  
getContent()  
getETags()  
getClientIP()
```

```
overrideGlobals()  
normalizeQuery()  
hasSession()  
getTrustedHosts()  
getUserInfo()  
isXmlHttpRequest()
```

Options de la classe Response

```
return new Response($contenu);
```

Au lieu de juste envoyer la réponse, il est possible de la construire élément par élément, notamment pour les entêtes HTTP.

La classe **Response** dispose de nombreux *helpers* pour gérer toutes les options et décomposer le processus d'envoi.

Types de réponses

Une réponse peut être un contenu envoyé au navigateur, mais également une redirection. Symfony offre trois possibilités :

```
render // renvoie une vue
redirect // renvoie vers un contrôleur (avec un code 302)
redirectToRoute // presque identique mais pointant vers une route
forward // délégation d'action (ne renvoie pas d'erreur HTTP)
```

Gestion des erreurs

Les erreurs peuvent se gérer de deux manières différentes :

- En modifiant le code d'erreur de la réponse

```
$reponse->setStatusCode(Response::HTTP_FORBIDDEN)
```

- En déclenchant une exception

```
throw new \Exception('Message d'erreur') \\ Erreur 500
throw this->createNotFoundException('Erreur') \\ Erreur 404
```

Pages d'erreurs

Lorsque qu'une erreur se produit, Symfony utilise un gabarit d'affichage particulier

- Dans l'environnement de production, il affiche juste un message d'erreur
- Dans l'environnement de développement, il affiche la trace complète de l'erreur

Pages d'erreurs

Il est possible de changer l'aspect de ces pages en modifiant (ou créant) le gabarit correspondant à l'erreur dans le dossier

```
/app/Resources/TwigBundle/views/Exception
```

Gestion des sessions

La classe **Request** offre un accès à une session utilisateur qui encapsule la superglobale `$_SESSION`

```
$session = $request->getSession();  
  
$session->get('param-1');  
  
$session->set('param-2', 'hello !');
```

Les messages Flash

Les messages Flash sont des messages destinés à s'afficher une seule fois dans le navigateur. Ils peuvent être stockés dans un objet de type Bag (FlashBag) et s'effacent automatiquement une fois utilisés.

Les messages Flash peuvent être triés par catégories et sont stockés dans la session utilisateur.

```
// dans le contrôleur  
$this->addFlash('warning', 'Aucun objet trouvé');  
  
$this->get('session')->getFlashBag();
```

Accès aux constantes de l'application

Le contrôleur peut également accéder aux constantes définies dans le fichier `parameters.yml`

```
// dans le contrôleur  
$this->getParameter('param-1');
```

Bonnes pratiques

- Créer une classe de contrôleur par fonctionnalité (services web, URL publiques, administration, rétro-compatibilité, etc.)
- Organiser des groupes de contrôleurs par dossier
- Ecrire des contrôleurs minces
- Ne pas utiliser les contrôleurs pour abriter du code métier ou du code lié à la présentation

Ce qu'on a vu

- Les détails des classes Request et Response
- La gestion des erreurs HTTP par les contrôleurs
 - et la personnalisation des pages d'erreur
- La gestion des sessions
- L'intégration de messages de notification
- Le lien avec les fichiers de configuration
- ... et quelques conseils d'organisation



Pour aller plus loin

Les rôles des contrôleurs est à la fois essentiel et circonscrit.

Ils peuvent se retrouver dans d'autres contextes comme des vues ou des services.

<http://api.symfony.com/3.0>



Alphorm.com

Twig

Introduction aux vues

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel Cadennes

Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Qu'est-ce que Twig ?
- Pourquoi utiliser Twig ?
- Les bases de la syntaxe
 - Variables
 - Directives
 - Filtres et fonctions
 - Commentaires
 - Héritage
- Conclusion
- Pour aller plus loin



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Qu'est-ce que Twig ?

- Twig est le « **moteur de template** » intégré à Symfony
- C'est un **pseudo-langage** de programmation
- Il sert à afficher les **vues** de l'architecture MVC
- Un composant développé par **Sensio Labs** pour Symfony, mais utilisé par d'autres plate-formes PHP
- Il est très bien intégré avec **Doctrine**

Pourquoi utiliser Twig ?

- Il permet de créer des hiérarchies de templates
- C'est une couche d'abstraction
- Il fait le lien avec le modèle et l'ORM (par défaut **Doctrine**)
- Il est très facilement extensible
- Et pourquoi pas directement PHP ?
 - une meilleure interface avec contrôleurs
 - une syntaxe plus « intuitive »
 - le mécanisme de cache

La syntaxe : les variables

- Les variables

```
{{ variable }}
```

```
{{ monObjet.prop1.slot235 }}
```

La syntaxe : Les directives

- Les directives

```
{% if ... %} ... {% endif %}
```

```
{% for x in liste... %}{% endfor %}
```

```
{% include... %}
```

La syntaxe : les commentaires

- Les commentaires

```
{# commentaires #}
```

La syntaxe : filtres et fonctions

- Les filtres et les fonctions

```
{{ variable | filtre }}
```

```
{{ fonction(param1, param2) }}
```

La syntaxe : hiérarchiser les squelettes

- L'héritage

```
{% extends... %}
```

```
{% block... %}{% endblock %}
```

N.B : la fonction **parent()** permet de conserver le contenu du parent

Ce qu'on a vu

- La syntaxe élémentaire de Twig
- Comment exploiter des données issues des contrôleurs
- Comment organiser et découper des templates



Pour aller plus loin

- Comment et pourquoi utiliser des **macros**
- Comment Twig dialogue avec Doctrine
- Comment étendre facilement les **filtres** et les **fonctions**
- Rendre la syntaxe **compatible** avec d'autres moteurs
- Embarquer un **contrôleur** dans le **template**
- Gérer les ressources (css, javascript, etc.) avec **Assetic**
- Faire du chargement **asynchrone**
- **Surcharger** un template



Alphorm.com

Twig

Inclure d'autres vues

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES

Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Embarquer des contenus externes
 - Inclure un autre squelette Twig
 - Embarquer un contrôleur secondaire
 - Chargement asynchrone des pages
- Autres directives
 - Intégrer correctement les URLs locales
 - Les variables globales
- Les ressources (assets)
 - Ajouter des ressources
 - Introduction à Assetic

Inclure une vue

- Inclure un squelette Twig dans un autre squelette se fait tout simplement par une directive include :

```
{% include <nomDuSquelette> %}
```

- Le nom du squelette est (par défaut) la notation raccourcie en trois segments (à ne pas confondre avec les contrôleurs) :

```
<nomDuBundle>:<dossierDeVue>:<nomDuFichier>
```

Embarquer un contrôleur

- Il est également possible de faire appel à un contrôleur depuis une vue :

```
{% render (controller(
    'AppBundle:Default:sousRequete',
    {'arg_1' : 0, 'arg_2': 'none' }
) %}
```

Charger une vue de manière asynchrone

- Symfony autorise le chargement asynchrone, qui peut être très utile lorsque les pages sont lourdes :

```
{{ render_hinclude(url(<URL>)) }}
```

```
{{ render_hinclude(controller(<contrôleur>)) }}
```

- Cette fonctionnalité oblige l'inclusion du script [hinclude.js](#) dans la page HTML

Charger une vue de manière asynchrone

- Il faut modifier la configuration de Symfony pour pouvoir exécuter de manière asynchrone des sous-contrôleurs
- Il est possible d'associer un affichage par défaut en attendant que le fragment soit chargé
- Cet affichage par défaut peut être défini pour toute l'application

Les liens

- Les liens internes peuvent avantageusement déclarés par le nom de leur route avec les directives path et url :

```
<a href='{ { path(<uneRoute>, {<arguments_json>}) } }'>Lien</a>
```

```
<a href='{ { url(<uneRoute>, {<arguments_json>}) } }'>Lien</a>
```

Variables globales

- Enfin, les squelettes ont directement accès à quelques variables globales de l'application :

```
app.user    // L'utilisateur courant
app.request // L'objet Request
app.session // Les variables de session
app.environment // L'environnement courant
app.debug   // Debug mode activé ?
```

Intégrer des ressources

Les ressources

- L'inclusion de fichiers de ressources se fait par la directive **asset**, donc le but principal est d'assurer une meilleur portabilité des applications.

```
<img src='{{ asset("images/logo.png") }}' />
```

Scripts et feuilles de styles

- Pour les scripts et les feuilles de style, on privilégiera plutôt des directives spécialisées, qui tireront par la suite parti des ressources du composant **Assetic**.

```
{% block stylesheets %}  
    {% stylesheets '<dossier_styles>/* %'  
        <link rel = 'stylesheet' href= '{{ asset_url }}' />  
    {endstylesheets %}  
{% endblock %}
```

Présentation d'Assetic

- Assetic est **un composant tiers** dont le but est de gérer les fichiers de ressources des applications PHP.
- Le principe est repris de la bibliothèque **webassets**, développée pour Python
- Cette formation n'entrera pas dans les détails d'Assetic mais entend juste présenter ses possibilités

Fonctionnalités d'Assetic

- **Minifier** des fichiers (javascript, CSS)
- **Rassembler plusieurs fichiers** dans un seul pour diminuer les requêtes
- **Compiler** des fichiers CSS à partir de sources **SASS**
- **Optimiser des images**
- Appliquer des « **filtres** » à des fichiers de ressources
- Mettre en cache des ressources

Fonctionnalités d'Assetic

- Assetic peut nécessiter l'installation de ressources système pour fonctionner, par exemple :
 - **SASS** (Ruby) pour activer le pré-processeur CSS
 - **JpegOptim** pour l'optimisation des images

Ce qu'on a vu

- Modulariser les squelettes en :
 - incluant des squelettes externes
 - calculant des contenus en faisant appel à un contrôleur
 - permettant le chargement asynchrone de blocs
- Gérer les ressources (js, css, images)
- Intégrer correctement des liens internes





Alphorm.com

Twig Etendre Twig

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES

Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Les macros
- Les filtres et les fonctions
- La classe \Twig_Extension
 - Créer une classe d'extension
 - Créer les méthodes correspondant aux filtres et fonctions
 - Déclarer l'extension au noyau de Symfony
- Utiliser l'extension
- Conclusion
- Pour aller plus loin



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Les macros

- Une macro est juste **une fonction définie** à l'intérieur du template.
- On la déclare avec la directive :

```
{% macro %}
```

- On peut importer des fichiers de macro avec la directive :

```
{% import 'mesMacros.html' as macros %}
```

La filtres et les fonctions

- Un **filtre** est communément un moyen de transformer la présentation d'une donnée.

```
{{ data | filtre }}
```

- Un filtre peut naturellement admettre des paramètres supplémentaires :

```
{{ \DateTime::now() | date('fr') }}
```

- La seule différence entre les filtres et les fonctions tient à l'échappement de la sortie

La filtres et les fonctions

- Une **fonction** est communément un moyen de calculer une valeur en fonctions de certaines données.

```
{{ maFonction(arg1, arg2) }}
```

La classe Twig_Extension

- **Filtres et fonctions** sont définis dans une classe d'extension qui doit hériter de la classe **Twig_Extension**, qui fait partie des composants de Symfony.
- Celle-ci oblige à définir (au moins) trois méthodes :
 - **getFilter** -> déclare la liste des filtres
 - **getFunctions** -> déclare la liste des fonctions
 - **getName** -> crée un nom (unique) pour la classe d'extension

Créer la classe d'extension

- Pour créer la classe d'extension, il faut en premier lieu la placer dans le dossier `Twig` d'un *bundle*. Et juste déclarer :

```
class mesExtensions extends \Twig_Extension
```

- A noter que `Twig_Extension` est dans l'espace de noms racine

Créer les filtres et les fonctions

- Par convention, les méthodes des filtres et des fonctions sont (comme souvent dans Symfony) suffixées.

- Ainsi : `function priceFilter ()` crée le filtre **price**.

- Et : `function totalFunction (x, y)` crée la fonction **total**.

Déclarer l'extension

- Pour pouvoir utiliser les méthodes d'extension, il faut encore les déclarer au noyau de Symfony dans le fichier **services.xml** du *bundle*.

```
<!-- monBundle/config/services.xml -->
<services>
    <service
id= "app.twig_extension"
class="AppBundle\Twig\AppExtension"
        public="false">
        <tag name="twig.extension" />
    </service>
</services>
```

Utiliser l'extension

- Une fois cela fait, les méthodes de la classe d'extension sont utilisables exactement comme celles fournies par défaut par Twig :

```
{{ data | filtre }}
```

```
{{ \DateTime::now() | date('fr') }}
```

Ce qu'on a vu

- Les extensions à la syntaxe de Twig : **macros, filtres et fonctions**
- Comment exploiter la classe `\Twig_Extension`
- Exploiter les nouveaux éléments syntaxiques pour créer des templates plus lisibles, modulaires et réutilisables



Alphorm.com

Les formulaires Introduction aux formulaires

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Rôle des formulaires
- Une méthode simple par la ligne de commande
 - Limites de cette méthode
- La classe **AbstractType**
- Anatomie d'une classe de formulaires
- Créer un formulaire simple *ex nihilo*



Le rôle des formulaires

- Au niveau de HTML, un formulaire est une collection de widgets qui sont soumis (submit) à une cible (action de l'élément form). Ces widgets ont depuis HTML5 pris beaucoup de variété
- Au niveau de PHP, un formulaire est transmis par une requête ayant un type MIME multipart/form-data, dont les données sont stockées dans la **variable superglobale \$_POST**
- Au niveau de Symfony, un formulaire est **une interface** qui doit être mise en correspondance avec **un modèle** (dans la plupart des cas)

Une méthode simple par la ligne de commande

- Symfony possède une commande qui permet de créer simplement et de manière interactive un formulaire

```
php app/console generate:doctrine:form <Bundle>:<Entity>
```

Limites

- Cette commande crée juste un squelette minimal de classe
- On ne peut pas prendre en compte les relations entre les entités du modèle

La classe AbstractType

- La classe **AbstractType** fournit quelques méthodes simples pour construire des classes de formulaires. Principalement :
 - **buildForm(\$form, \$options)** : méthode chargée de construire le formulaire
 - **configureOptions(\$resolver)** : méthode permettant de paramétrer le formulaire

La classe AbstractType

- La classe AbstractType fournit aussi deux méthodes pour construire le rendu du formulaire :

```
buildView(FormView $view, FormInterface $form, array $options)
finishView(FormView $view, FormInterface $form, array $options)
```

Création d'une classe de formulaire simple

- La méthode la plus simple pour créer un formulaire consiste bien souvent à utiliser la ligne de commande pour créer le squelette de base. Puis à compléter la structure engendrée avec les informations manquantes.
- D'autant que nous verrons que, dans les applications réelles, les formulaires sont souvent plus complexes

add

- Définir un formulaire consiste la plupart du temps à ajouter des champs au constructeur.

```
$builder
->add('libelle')
->add('categorie', null, array('placeholder' => 'Catégorie'))
->add('dateLimite', DateType::class, array('widget' => 'single_text'))
->add('save', SubmitType::class)
```

Options

- **FormBuilderInterface** définit de nombreuses méthodes pour les constructeurs de formulaires. Parmi elles :

```
$builder  
    ->setMethod($options['method'])  
    ->setAction($options['action'])
```

- **\$options** fait référence ici à l'argument de la méthode buildForm

data_class

- Bien que pas toujours requis, il est de bonne pratique d'indiquer au formulaire quelle est l'entité associée. Ceci se fait dans **configureOptions** :

```
public function configureOptions(OptionsResolver $resolver)  
{  
    $resolver->setDefaults(array(  
        'data_class' => '<nomDuBundle>\Entity\<nomEntite>',  
    ));  
}
```

Ce qu'on a vu

- La représentation déclarative des formulaires dans Symfony
- Comment créer des formulaires par défaut
- La classe AbstractType sous-jacente aux formulaires
- Un formulaire simple



Alphorm.com

Les formulaires
Rendu et utilisation
des formulaires

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Les champs de formulaire
- Afficher les champs dans une vue
 - Personnaliser l'aspect des champs
- Le cycle du formulaire
 - Intégration dans le contrôleur
 - Traitement par Doctrine
 - Récupérer les erreurs
- Validation des formulaires



Les champs d'un formulaire

Les champs de formulaire

- Il existe de nombreux types de champs. Toutes ces classes sont des enfants de la classe **FormType** :
 - **Texte** : `TexteType`, `EmailType`, `MoneyType`, etc.
 - **Choice** : `ChoiceType`, `CountryType`, `EntityType`, etc.
 - **Date** : `DateType`, `TimeType`, `BirthdayType`
 - **Divers** : `CheckboxType`, `RadioType`, `FileType`, `HiddenType`
 - **Groupe**s : `CollectionType`, `RepeatedType`

Une classe de champ

- Chaque objet champ est associé à :
 - Un widget HTML pour le rendu dans la page
 - Un ensemble d'options qui lui sont propres
 - Un ensemble d'options héritées de ses parents

Les fonctions de formulaires pour Twig

```
{{ form(form, vars) }}
```

```
{{ form_start(form, vars) }}  
{{ form_end(form) }}
```

Les fonctions de formulaires pour Twig

```
{{ form_widget(form, vars) }}
```

```
{{ form_label(form, label, vars) }}  
{{ form_row(form, vars) }}  
{{ form_errors(form, vars) }}
```

```
{{ form_rest(form, vars) }}
```

Personnaliser l'aspect des champs

- Il est également possible de créer ses propres gabarits d'affichage pour des types de champs existants.
- Pour cela, il suffit de :
 - A. Créer un gabarit dans le dossier **views** d'un bundle
 - B. Déclarer le gabarit à **Twig**
 - C. Importer le nouveau gabarit dans le **squelette** de la page

Utilisation des formulaires

Le cycle du formulaire

- Intégrer un formulaire dans un contrôleur se fait en deux temps :

```
$form = $this->createForm(TaskType::class, $task, $options);  
... code ...  
form->createView();
```

Le traitement par Doctrine

- Quand le contrôleur reçoit le formulaire rempli, il faut le faire persister.
Pour cela :
 - **handleRequest()** : hydrate un objet avec les données du formulaire
 - **persist()** : déclare à Doctrine qu'un objet doit être pris en compte
 - **flush()** : modifie le modèle

Récupérer les erreurs

- Dans le navigateur par la configuration des options
- Par des classes de validation associées au formulaire
- Par la détection de Symfony si rien n'est spécifié

Ce qu'on a vu

- Les champs de formulaire
- Afficher et Personnaliser l'aspect des champs
- Le cycle de traitement des formulaires par Symfony
- La gestion des erreurs





Alphorm.com

Les formulaires Formulaires complexes

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Des formulaires plus complexes
 - Créer un type de champ de formulaire
- Associer des modèles ad hoc à un formulaire
- Héritage de formulaire
- Validation des données du formulaire



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Formulaires complexes

Les formulaires enfants

- Dans bon nombre cas, les formulaires que l'on a affiché font référence à plusieurs entités, car ces entités sont liées.
- Symfony accepte que vous embarquiez un formulaire complet à l'intérieur d'un autre formulaire.

```
$builder->add('categorie', CategoryType::class)
```

- Et pour afficher les champs correspondants :

```
{{ form_row(form.categorie.nom) }}
```

Les collections de formulaires

- Si une entité est liée à plusieurs autres entités d'une même classe (e.g. une personne qui a plusieurs adresses), il est possible d'utiliser un type de champ **CollectionType** pour laisser Symfony gérer les associations automatiquement.

```
$builder->add('adresses', CollectionType::class, array(
    'entry_type' => AdresseType::class
));
```

- Il suffit ensuite d'une boucle dans le **squelette Twig** pour afficher les sous-formulaires associés.

Les collections de formulaires

- Lors du traitement du formulaire, il faudra, selon les cas, faire attention au sens directeur de l'association si l'on veut que la persistance soit prise en charge de manière transparente.
- Symfony met à disposition une option **allow_add** qui autorise, moyennant un fragment de code JavaScript, à ajouter des sous-formulaires à la volée à la collection existante.
- Symfony met également une option **allow_delete** pour l'effet inverse.

Types de champs personnalisés

- Définir un type de champ personnalisé n'est pas très différent de définir un formulaire. Il faut :
 - Créer une classe de formulaire et indiquer que cette classe est une sous-classe de **FormType**
 - Créer le gabarit d'affichage associé au champ
 - Déclarer le gabarit au noyau de Symfony

Les modèles de formulaire

Les modeles de formulaire

- Un autre cas de figure est celui où l'on a besoin de champs supplémentaires qui ne sont pas dans le modèle.
- Il est alors possible de créer une entité qui sera hébergée dans un dossier **Model** du dossier **Form**, et de s'en servir dans l'application.

L'héritage de formulaire

Héritage de formulaire

- Comme les formulaires sont des classes, un bénéfice collatéral est de pouvoir faire fonctionner le mécanisme d'héritage, ce qui s'avère particulièrement utile si vous avez des entités organisées hiérarchiquement :

```
class CategoryType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        parent::buildForm($builder, $options);
    }
}
```

La validation de formulaire

Validation de formulaire

- Pour valider des données de formulaire, la solution la plus simple est de passer par **les annotations**.
- Pour cela, il faut commencer de mettre à jour le fichier **config.yml** :

```
# app/config/config.yml
framework:
    validation: { enable_annotations: true }
```

Validation de formulaire

- Ensuite, il n'y plus qu'à annoter le fichier d'entité

```
class Personne
{
    /**
     * @Assert\Choice(
     *     choices = { "masculin", "féminin", "autre" },
     *     message = "Chosissez un genre vallide."
     * )
     */
    public $sexe;
```

Validation de formulaire

- Il y a des classes de validation pour un grands nombre de propriétés des variables : chaînes de caractères, dates, expressions régulières, collections, fichiers.
- Les mutateurs (getters) associés à des propriétés protégées ou privées peuvent également être validées
- En dehors des annotations, les fichiers de validation doivent être hébergés dans le répertoire :

Resources/config/validation

Groupes de validation

- Selon les cas, on peut souhaiter restreindre la validation à certains champs seulement du formulaire. Dans ce cas on peut définir des groupes de validation.

```
/**
 * @Assert\Email(groups={"registration"})
 */
private $email;
```

- Il suffit ensuite de déclarer le groupe considéré dans les options du formulaire.

Ce qu'on a vu

- Créer des formulaires plus complexes
- Créer ses propres types de champs
- Créer des modèles de formulaire *ad hoc*
- Hiérarchiser les classes de formulaire
- Valider les données des formulaires



Alphorm.com

Les modèles Introduction

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Qu'est-ce qu'un modèle ?
- Construire un modèle conceptuel
- Dualité des modèles : objets et relations
- Introduction aux ORM



Le modèle conceptuel

- Créer un modèle conceptuel est le travail qui consiste à recenser et organiser toutes les informations relatives à un domaine donné pour en construire **une représentation structurée**.
- Ces informations sont regroupées de manière à respecter la sémantique du domaine au sein d'**entités**
- Les entités sont organisées entre elles selon des **relations significantes**
- Le modèle conceptuel est à priori **indépendant** de toute réalisation technique

Construire un modèle conceptuel

- Des langages de représentation permettent de traduire les informations relatives à un domaine en un modèle opérationnel Entité - Relation.
- Une méthode comme **Merise** possède son propre langage
- **UML** est un formalisme qui développé parallèlement aux méthodes itératives
- **RDF, RDFS, OWL** sont des langages utilisés dans le domaine du Knowledge Engineering

Symphony et les modèles

- Symphony est conçu de manière à alléger les tâches de gestion administrative des données
- Une vision architecturale de l'application est privilégiée
- Plus le travail est préparé en amont de l'implémentation, plus la partie technique sera facilitée et automatisable.

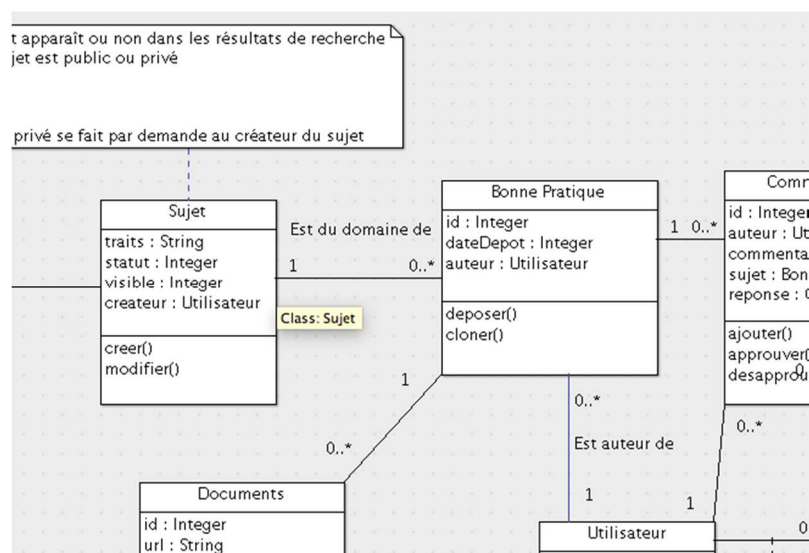
UML

UML

- UML est un **langage de représentation** qui définit des « vues » alternatives et complémentaires sur l'application et a pour vocation d'accompagner le développement depuis la conception jusqu'au déploiement.
- Nous nous intéresserons ici aux **diagrammes de classes**

Schéma UML

- Les diagrammes de classes de UML peuvent représenter (principalement) :
 - Des classes (propriétés et méthodes)
 - Des associations (avec leur arité)
 - Des agrégations de classes
 - Des hiérarchies de classes
 - Des packages (regroupement logiques)



Export du modèle

- UML a la possibilité de produire des squelettes de classes dans différents langages dont PHP
- Il est donc possible de se servir d'un outil de modélisation comme fournisseur de matière première pour la construction d'une application sous Symfony

Les modèles et ORM

Objets et relations

- Le paradigme des la POO s'est généralisé à l'immense majorité des langages de programmation
- Le modèle objet ne s'est toutefois pas posé dans le monde des bases de données
- Restent trois alternatives :
 - Utiliser des outils comme **PDO** pour dialoguer avec des bases de données relationnelles
 - Utiliser une couche d'abstraction permettant l'interrogation transparente de bases de données relationnelles
 - Utiliser des bases **NoSQL** ou graphes (**MongoDB**, etc.)

ORM : *Object Relational Mapper*

- Symfony ne tranche pas définitivement entre ces alternatives.
- Par défaut, il est prévu pour fonctionner avec un **ORM** (couche d'abstraction) Objet/Relationnel : **Doctrine**
- Il est possible d'utiliser d'autres ORM, comme **Propel**, par exemple.

Rôle de l'ORM

- Comme couche d'abstraction, l'ORM est capable de :
 - Transformer un objet complexe en un jeu de données adéquat pour un ensemble de relations (au sens SQL).
 - Utiliser des données issues de tables de la base de données pour « hydrater » des objets qui leur correspondent
- Son rôle est aussi d'assurer la persistance des données, c'est-à-dire de veiller à la cohérence entre l'état du programme et celui du modèle.

Coûts des ORM

- Un ORM apporte énormément de confort pour le développement d'applications.
- En revanche, cela a un coût en matière de **lourdeur** des applications
- Il est généralement conseillé d'installer des gestionnaires de cache comme **memcache** et/ou APC

Ce qu'on a vu

- Pourquoi un bon modèle est essentiel
- Comment préparer un modèle conceptuel en amont d'une application sous Symfony
- Le rôle des ORM



Alphorm.com

Les modèles Construction du modèle

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Passer du modèle conceptuel au modèle de données
 - Les outils de Symfony
- Organisation des classes de modèles dans Symfony
 - Les entités
 - Les « repositories »
- Travailler avec les modèles dans l'application
 - « Persister » les données
 - Le mode transactionnel



Introduction

- Passer du modèle conceptuel à un modèle concret utilisable dans Symfony suppose plusieurs opérations :
 1. La création d'un **modèle déclaratif** de l'ensemble des classes
 2. La création de **classes d'entités** au sens PHP5
 3. La création du **schéma** de base de données correspondant

Les stratégies

- Il y a encore trois possibilités pour arriver à ce résultat :
 1. Partir de la base de données pour créer les entités
 - > *practicable en cas de reprise d'un site existant*
 2. Partir des classes d'entités pour engendrer la base de données
 - > *possible à condition de travailler avec des annotations*
 - > *permet de repartir de squelettes engendrés par des outils UML*
 3. Ecrire le schéma déclaratif puis engendrer les entités et la base de données
 - > *le schéma habituel*
 - > *avec la possibilité d'utiliser la console de Symfony*

Les entités

Architecture des entités

- Toutes les entités de l'application correspondent à une classe hébergée dans le dossier :



- Les classes d'entités sont généralement accompagnées d'une classe de requête (repository) qui porte le nom de l'entité suffixée par **Repository** :

```
<Entite>Repository.php
```

Architecture des entités

- Par ailleurs, toutes les entités doivent être déclarées à Doctrine. Elles peuvent l'être sous deux formes :
 - Par des annotations incluses dans le fichier de la classe d'entité
 - Par un fichier de déclaration hébergé dans le dossier :

```
Resources/config/doctrine
```

Un déclaration simple

```
<doctrine-mapping [...] />
  <entity name='...\Entity\Article'
    repository-class='...\Entity\ArticleRepository'>

  <id name='id' type='integer' column='id'>
    <generator strategy = "AUTO" />
  </id>

  <field name="corps" type="text" column="corps" />
  <field name="sponsor" type="boolean" column="sponsor" default="0" />
  </entity>
</doctrine-mapping>
```

La classe de requêtes

- A toute entité peut être associée **une classe de requêtes** permettant d'explorer le modèle.
- Ces classes héritent toutes, directement ou indirectement, de la classe :

`EntityRepository`

- La syntaxe des requêtes fera l'objet d'une session particulière.

Les accesseurs

- Les classes de requêtes disposent d'emblée de **plusieurs méthodes** pour extraire des données du modèle :
- **find(\$id)**
// trouve une entité en fonction de la valeur de la clef primaire
- **findOneBy(\$critere)**
// trouve la première entité correspondant critère
- **findBy(\$criteres, \$tri, \$limite, \$offset)**
// trouve les entités dont la propriété \$clef correspond à \$valeur
- **findOneBy<Clef>(\$valeur)**
// raccourci syntaxique de la méthode précédente
- **findBy<Clef>(\$valeur)**
// raccourci syntaxique de la méthode précédente
- **findAll()**
// trouve toutes les entités (fortement déconseillé)

Travailler avec les entités

Entity Manager

- Dans l'architecture MVC, ce sont principalement **les contrôleurs** qui accèdent au modèle.
- Dans Symfony, cet accès se fait **via un service** :

```
'doctrine.orm.entity_manager'
```

Entity Manager

- L'Entity Manager a pour rôle :
 - de gérer les requêtes au modèle (via les classes de requêtes)
 - de gérer la persistance des données, c'est-à-dire assurer la sauvegarde et la restauration des données, ainsi que la synchronisation entre l'état du programme et le modèle.

Entity Manager : requêtes

- L'Entity Manager permet l'accès à la **classe de requêtes** de l'entité :

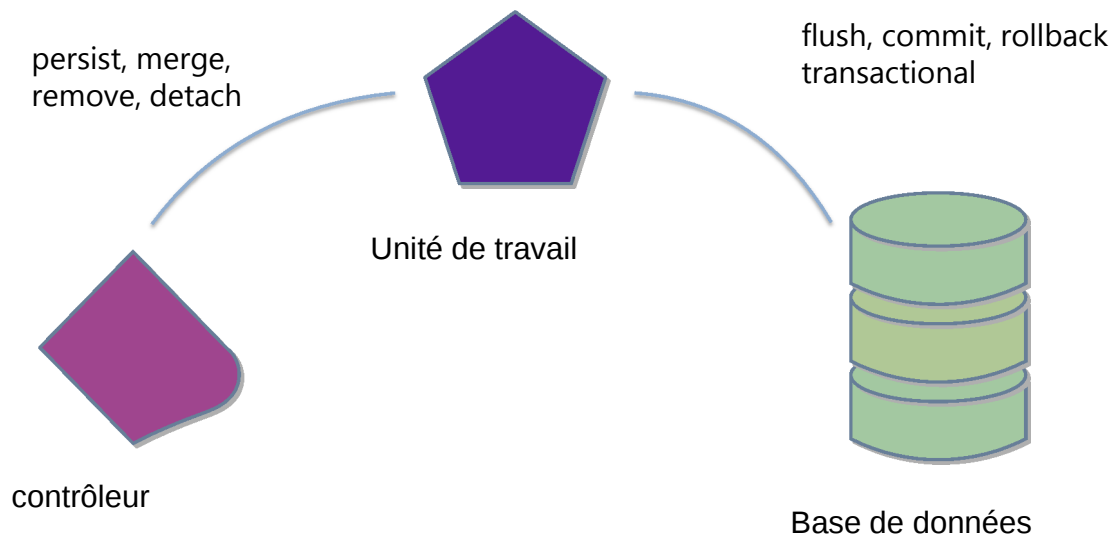
```
$repo = $em->getRepository(<Entite>)
```

- C'est ensuite le **Repository** qui prend en charge les requêtes :

```
$repo->maRequete(<arguments>)
```

La persistance

Le mécanisme de la persistance



Le mécanisme de la persistance

- L'Unité de Travail (*UnitOfWork*) est chargée de maintenir à jour toutes les tâches à réaliser pour maintenir la synchronisation entre état du programme et la base de données
- Chaque entité devant être ajoutée au modèle ou modifiée doit être déclarée à l'Unité de Travail (**persist**)
- Les **entités extraites** du modèle sont automatiquement déclarées à l'Unité de Travail
- Les entités sont manipulées dans l'Unité de Travail jusqu'au moment où le modèle est remis à jour (**flush**)

Persister

persist

- La méthode **persist** n'a à être appelée explicitement que lors de la création d'un objet.
- Elle ajoute cet objet à la liste des entités à ajouter au modèle.

```
$em->persist($entity)
```

remove

- La méthode **remove** indique à l'Unité de Travail que l'entité considérée devra être ôtée du modèle lors de la prochaine synchronisation.

```
$em->remove($entity)
```

detach

- La méthode **detach** permet de retirer une entité de la liste des entités à synchroniser par l'Unité de Travail

```
$em->detach($entity)
```

merge

- La méthode **merge** permet de réassigner à l'Unité de Travail une entité à prendre en charge (par exemple, parce qu'elle a été détachée ou sérialisée)
- Pour une nouvelle entité, **merge** est équivalent de **persist**.

```
$entity = $cacheDriver->fetch($cacheId);  
// Exemple : restauration d'un objet depuis un cache  
$em->merge($entity)
```

clear

- La méthode **clear** vide l'Unité de Travail.

```
$em->clear()
```

La synchronisation

flush

- La méthode la plus couramment utilisée est **flush**.
- Elle inspecte toutes les entités contenues dans l'Unité de Travail et met à jour la base de données.
- **flush** travaille implicitement dans un mode « **auto-commit** »

```
$em->flush()
```

commit / rollback / transactional

- Il est possible de gérer manuellement la mise à jour du modèle en utilisant **commit** pour procéder à la mise à jour et **rollback** pour la restauration en cas d'erreur.
- **Doctrine** fournit une méthode transactionnelle qui encapsule la gestion de la restauration :

```
try {  
    ... code ...  
    $transaction->commit();  
} catch (Exception $e) {  
    $transaction->rollback();  
}
```

```
$em->transactional(  
    function ($em) {  
        ... code ...  
    }  
)
```

Ce qu'on a vu

- Comment passer d'un modèle conceptuel à un schéma de base de données
- Comment créer des entités représentant le modèle de données de l'application
- Quelle est l'architecture des entités dans Symfony
- Le rôle du moteur de **persistance**
- Le fonctionnement du moteur par défaut de Symfony : **Doctrine**





Alphorm.com

Les modèles La persistance

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Les mécanismes de persistance
- La gestion de l'Unité de Travail
- La synchronisation du modèle
- Le mode transactionnel

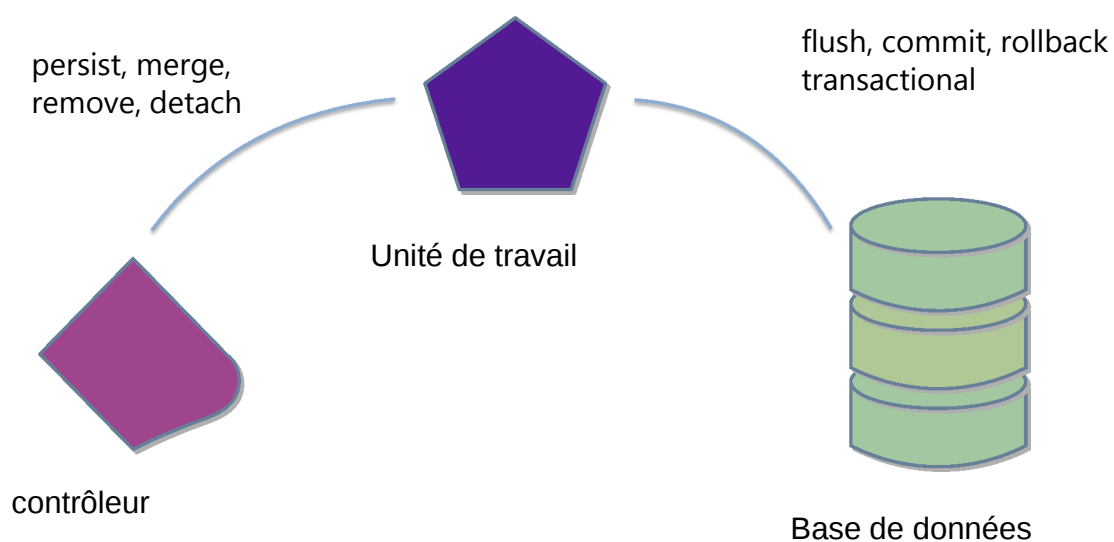


Formation Symfony3, les fondamentaux

alphorm.com™ ©

La persistance

Le mécanisme de la persistance



Le mécanisme de la persistance

- L'Unité de Travail (UnitOfWork) est chargée de maintenir à jour toutes les tâches à réaliser pour maintenir la synchronisation entre état du programme et la base de données
- Chaque entité devant être ajoutée au modèle ou modifiée doit être déclarée à l'Unité de Travail (persist)
- Les entités extraites du modèle sont automatiquement déclarées à l'Unité de Travail
- Les entités sont manipulées dans l'Unité de Travail jusqu'au moment où le modèle est remis à jour (flush)



Persister

persist

- La méthode persist n'a à être appelée explicitement que lors de la création d'un objet.
- Elle ajoute cet objet à la liste des entités à ajouter au modèle.

```
$em->persist($entity)
```

remove

- La méthode remove indique à l'Unité de Travail que l'entité considérée devra être ôtée du modèle lors de la prochaine synchronisation.

```
$em->remove($entity)
```

detach

- La méthode detach permet de retirer une entité de la liste des entités à synchroniser par l'Unité de Travail

```
$em->detach($entity)
```

merge

- La méthode merge permet de réassigner à l'Unité de Travail une entité à prendre en charge (par exemple, parce qu'elle a été détachée ou sérialisée)
- Pour une nouvelle entité, merge est équivalent de persist.

```
$entity = $cacheDriver->fetch($cacheId);  
// Exemple : restauration d'un objet depuis un cache  
$em->merge($entity)
```

clear

- La méthode clear vide l'Unité de Travail.

```
$em->clear()
```

La synchronisation

flush

- La méthode la plus couramment utilisée est flush.
- Elle inspecte toutes les entités contenues dans l'Unité de Travail et met à jour la base de données.
- flush travaille implicitement dans un mode « auto-commit »

```
$em->flush()
```

commit / rollback / transactional

- Il est possible de gérer manuellement la mise à jour du modèle en utilisant commit pour procéder à la mise à jour et rollback pour la restauration en cas d'erreur.
- Doctrine fournit une méthode transactionnelle qui encapsule la gestion de la restauration :

```
try {  
    ... code ...  
    $transaction->commit();  
} catch (Exception $e) {  
    $transaction->rollback();  
}
```

```
$em->transactional(  
    function ($em) {  
        ... code ...  
    }  
)
```

Ce qu'on a vu

- Ce que c'est que la persistance
- Le rôle de doctrine comme moteur de persistance
- Les méthodes de synchronisation du modèle et de l'état de l'application



Alphorm.com

Les modèles
Les associations

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Les associations
 - ManyToOne / OneToMany
 - OneToOne
 - ManyToMany
 - Relations uni- ou bi-directionnelles
 - Importance de la direction
- Hiérarchies d'entités

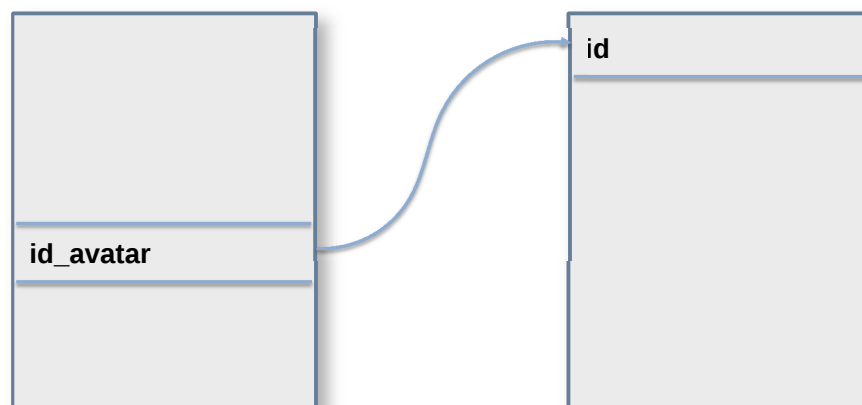


Direction

- Les **associations** entre entités peuvent **uni-directionnelles** ou **bi-directionnelles**
- En fonction de la déclaration, chaque entité à chaque bout de l'association définira ou non **une propriété** dénotant l'autre entité
- L'uni-directionnalité et **par défaut**.
- La bi-directionnalité s'indique par les attributs **inversed-by** et **mapped-by**

One to one

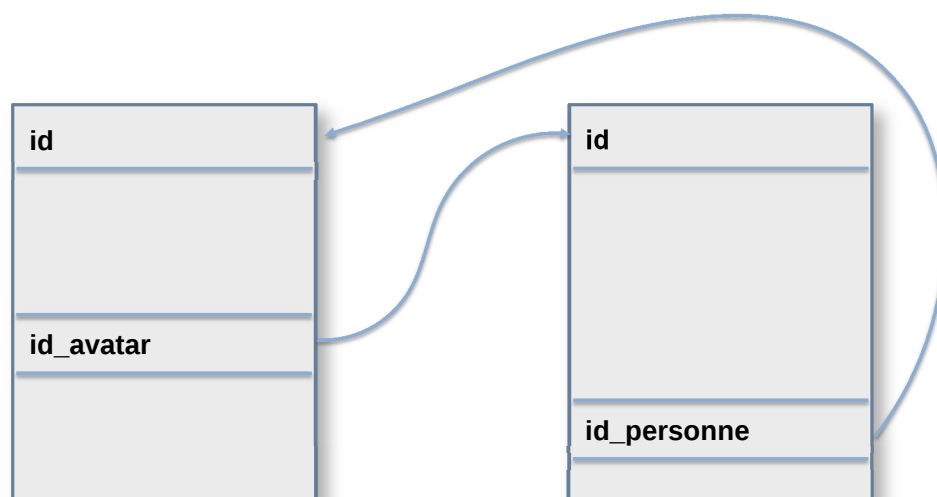
Exemple



One to One unidirectionnel

```
<doctrine-mapping>
  <entity class="Product">
    <one-to-one field="shipping" target-entity="Shipping">
      <join-column name="shipping_id" referenced-column-name="id" />
    </one-to-one>
  </entity>
</doctrine-mapping>
```

Exemple



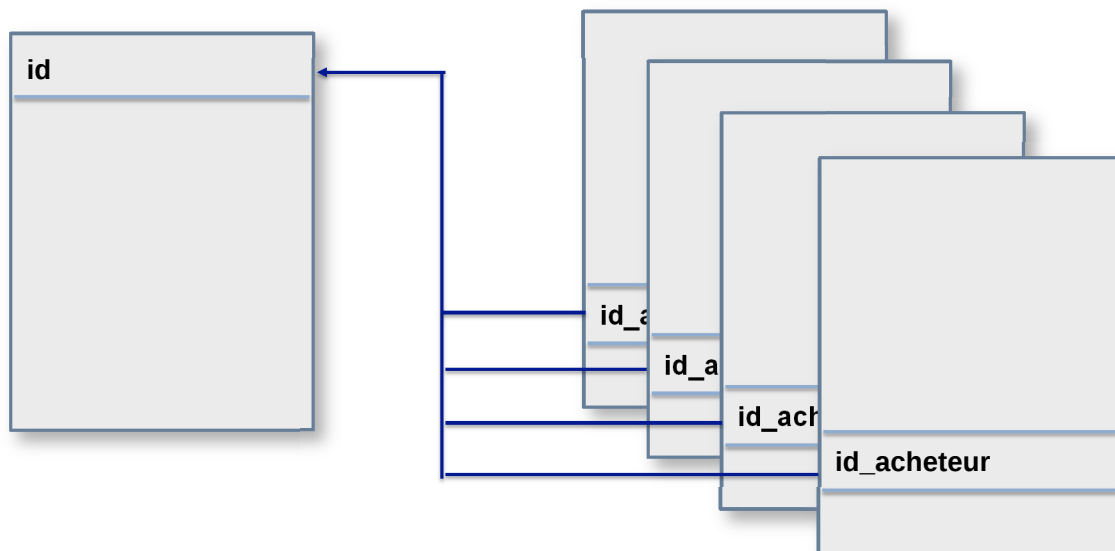
One to One bidirectional

```
<doctrine-mapping>
  <entity name="Customer">
    <one-to-one field="cart" target-entity="Cart" mapped-by="customer" />
  </entity>

  <entity name="Cart">
    <one-to-one field="customer" target-entity="Customer" inversed-by="cart">
      <join-column name="customer_id" referenced-column-name="id" />
    </one-to-one>
  </entity>
</doctrine-mapping>
```

Many to one

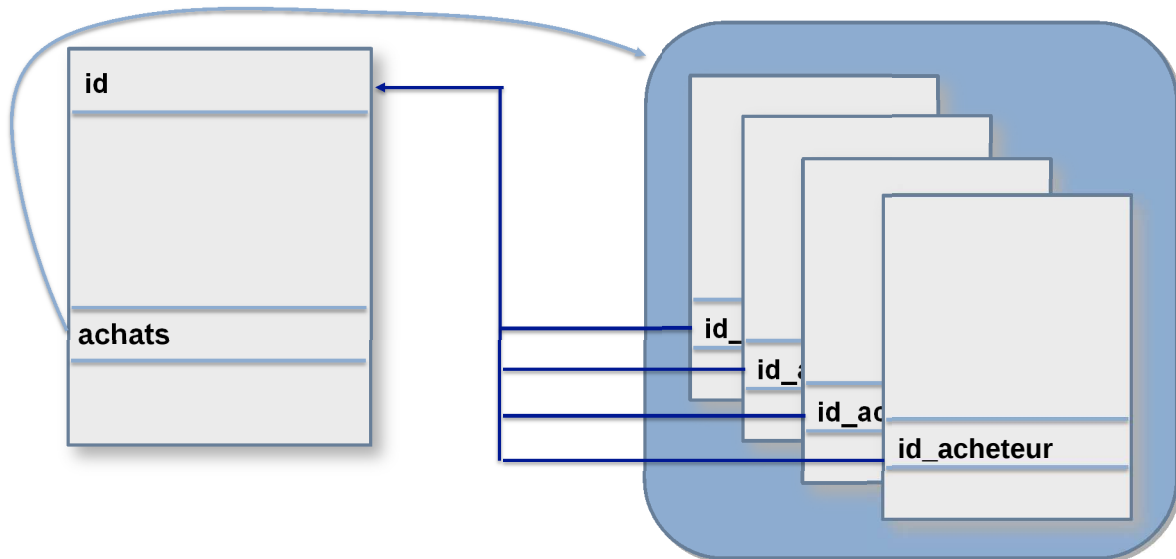
Exemple



Many To One unidirectionnel

```
<doctrine-mapping>
  <entity name="User">
    <many-to-one field="address" target-entity="Address">
      <join-column name="address_id" referenced-column-name="id" />
    </many-to-one>
  </entity>
</doctrine-mapping>
```

Exemple

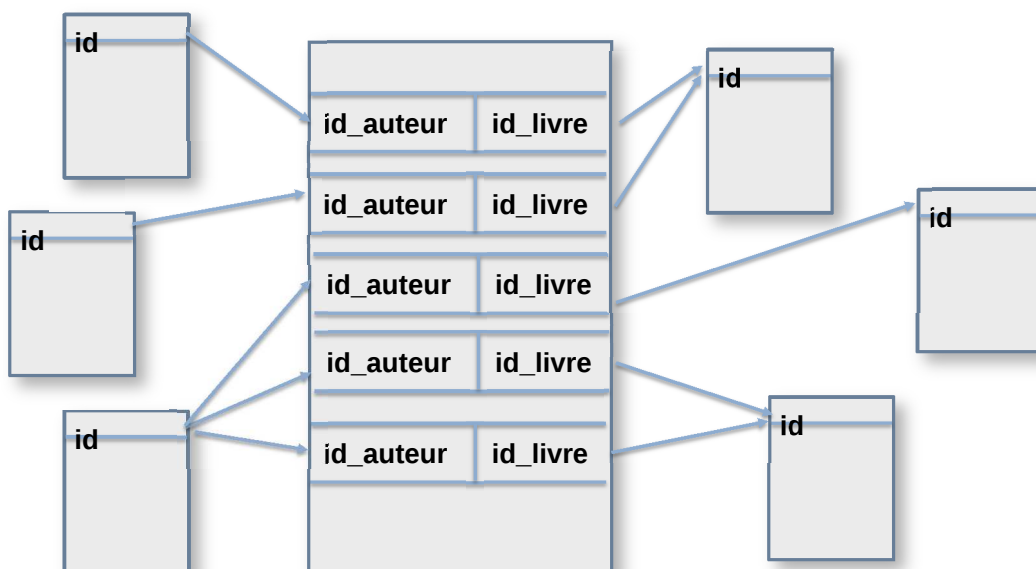


Many To One bidirectionnel (+ One To Many)

```
<doctrine-mapping>
  <entity name="Product">
    <one-to-many field="features" target-entity="Feature" mapped-by="product" />
  </entity>
  <entity name="Feature">
    <many-to-one field="product" target-entity="Product" inversed-by="features">
      <join-column name="product_id" referenced-column-name="id" />
    </many-to-one>
  </entity>
</doctrine-mapping>
```

Many to many

Exemple



Many To Many bidirectionnel

```
<doctrine-mapping>
  <entity name="User">
    <many-to-many field="groups" inversed-by="users" target-entity="Group">
      <join-table name="users_groups">
        <join-columns>
          <join-column name="user_id" referenced-column-name="id" />
        </join-columns>
        <inverse-join-columns>
          <join-column name="group_id" referenced-column-name="id" />
        </inverse-join-columns>
      </join-table>
    </many-to-many>
  </entity>

  <entity name="Group">
    <many-to-many field="users" mapped-by="groups" target-entity="User"/>
  </entity>
</doctrine-mapping>
```

Héritage d'entités

Héritage d'entités

- Le modèle **Doctrine** peut refléter la hiérarchie des classes du modèle conceptuel de l'application.
- Dans ce cas, il y a deux choses à faire :
 1. Déclarer la liste des entités-enfants dans l'entité-parent
 2. Supprimer la clef primaire de l'entité-enfant

Transformation de l'entité-parent

```
<entity name='Document' inheritance-type="JOINED" >
  <id name="idDocument" type="integer" column="id">
    <generator strategy = "AUTO" />
  </id>

  <discriminator-column name="typeDocument" type="string" />

  <discriminator-map>
    <discriminator-mapping value="livre" class="Livre" />
    <discriminator-mapping value="revue" class="Revue" />
  </discriminator-map>
</entity>
```

En pratique

- Dans la pratique, on suit le processus suivant :
 - Création du schéma des entités
 - Engendrement des classes d'entités
 - Ajout de l'héritage dans les classes d'entités produites
 - Création du schéma de la base de données

En pratique

- N.B. 1 : L'entité-parent joue le rôle d'une classe abstraite, bien que la classe correspondante ne soit pas nécessairement définie comme telle
- N.B. 2 : La classe parent permet de mutualiser un grand nombre de méthodes communes - toutes les classe enfants.

Ce qu'on a vu

- Les différentes sortes d'associations entre entités
 - L'importance de la direction de l'association
- La mise en œuvre de l'héritage entre entités dans le modèle Doctrine
 - Les stratégies de gestion des tables SQL



Alphorm.com

Les modèles
Les langages
de requêtes

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Classes de requêtes
- QueryBuilder
 - L'objet QueryBuilder
 - Récupérer les résultats
 - Ecrire des requêtes complexes



Les classes de requêtes

Classes de requêtes

- Toutes les requêtes se trouvent dans les classes de « **Repository** » (ou dépôt) associées aux entités
- Ces requêtes concernent principalement l'extraction de données du modèle. L'ajout, la modification et la suppression étant en fait gérés directement à partir du contrôleur
- Les classes de requêtes sont utiles pour les requêtes complexes.
- **Doctrine** admet 3 syntaxes différentes pour exprimer ces requêtes

Classes de requêtes

- Les classes de requêtes sont en général créées automatiquement lors de la création des entités
- Elles suivent une règle de nommage habituelle dans Symfony

```
Classe d'entité      : Auteur.php  
Classe de requête   : AuteurRepository.php
```

Les classes de requêtes

- Les contrôleurs possèdent un accès direct aux classes de requêtes via le service **Entity Manager**

```
em = $this->get('doctrine.orm.entity_manager');  
$repository = $em->getRepository('<Bundle>\Entity\<Entite>');
```

L'objet querybuilder

QueryBuilder

- **QueryBuilder** fournit une **API** pour engendrer des requêtes SQL à partir d'un langage objet.
- Elle est initialisée par l'instruction :

```
$qb = $this->createQueryBuilder('u');
```

N.B. : le paramètre 'u' est totalement arbitraire, la requête étant automatiquement liée à l'entité à laquelle fait référence la classe

L'objet QueryBuilder

- Une fois l'objet instancié, il peut construire trois types de requêtes :

```
$qb->select($champs);  
$qb->delete($entite);  
$qb->update($entite);
```

- Une requête de sélection peut être construite itérativement :

```
$qb->addSelect($champs);
```

L'API QueryBuilder

- join
- innerJoin
- leftJoin
- where
- orWhere
- andWhere
- groupBy
- addGroupBy
- having
- andHaving
- orHaving
- orderBy
- addOrderBy

Exemple

```
$qb->select('u')  
->where('u.id >= 25')  
->andWhere('u.id < 35')  
->orderBy('u.name', 'ASC')
```

N.B.: Les requêtes ne manipulent que les propriétés des objets et jamais les champs de la base de données

Les paramètres

- Pour rendre l'insertion de variables dans les requêtes, on a recours à la méthode :

```
$qb->setParameters(array(1 => '25', 2 => '35'));
```

```
$qb->setParameter('minimum', 25)  
->setParameter('maximum', 35) ;
```

Exemple

```
$qb->select('u')  
->where('u.id >= ?1')  
->andWhere('u.id < ?2')  
->orderBy('u.name', 'ASC')  
->setParameters(  
    array(1 => '25', 2 => '35')  
);
```

```
$qb->select('u')  
->where('u.id >= :minimum')  
->andWhere('u.id < :maximum')  
->orderBy('u.name', 'ASC')  
->setParameter('minimum', 25)  
->setParameter('maximum', 35) ;
```

Limiter les résultats

- Pour prendre juste un segment de l'ensemble des résultats :

```
$offset = (int)$_GET['offset'];  
$limit = (int)$_GET['limit'];  
  
$qb->setFirstResult( $offset )  
    ->setMaxResults( $limit );
```

Exécuter la requête

- Une fois la requête construite, il ne reste plus qu'à engendrer le SQL correspondant, ce qui est fait par :

```
$query = $qb->getQuery()
```

- Et récupérer les résultats :

```
$results = $query->getResults()
```



Les résultats

N.B. : Doctrine rend une collection d'objets, instances des entités déclarées dans la requête

Les fonctions selectives

La classe Expr

- Doctrine peut exprimer un sous-ensemble (restreint) des fonctions disponibles en SQL au travers de la classe **Expr**.

```
$expression = $qb->expr();
$orExpression = $expression->orX();
$qb->select('u')
    ->where(
    $orExpression(
        $expression->eq('u.id', '?1'),
        $expression->like('u.nickname', '?2')
    ))
```

Conclusion



- QueryBuilder est une **API** de haut niveau
- Elle permet de faire complètement **abstraction de la couche SQL**
- Elle permet de **construire les requêtes** bloc par bloc
- Elle laisse Doctrine **optimiser le SQL** produit
- Elle ne couvre qu'un **sous-ensemble de SQL**
- Elle est naturellement **plus lente** que du SQL natif



Ce qu'on a vu

- Le lien organique entre les classes d'entités et les classes de requêtes
- Comment construire une requête avec l'objet QueryBuilder
- Les avantages et les inconvénients de cette méthode





Alphorm.com

Les modèles DQL

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES

Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Présentation de DQL
- Les requêtes avec DCL
- Fonctions, opérateurs, agrégats



Formation Symfony3, les fondamentaux

alphorm.com™ ©

DQL

Présentation

- DQL est l'acronyme de **Doctrine Query Language**
- C'est un langage assez similaire à **Hibernate** dans l'univers Java, voire **SPARQL** pour le web sémantique.
- C'est un langage utilisant une syntaxe SQL, mais appliquée à des objets (entités de Symfony) et non des champs/tables SQL
- Doctrine produit le SQL associé et l'optimise

Classes de requêtes

- Les classes de requêtes sont en général créées automatiquement lors de la création des entités
- Elles suivent une règle de nommage habituelle dans Symfony

Classe d'entité : Auteur.php
Classe de requête : AuteurRepository.php

Les classes de requêtes

- Les contrôleurs possèdent un accès direct aux classes de requêtes via le service Entity Manager

```
em = $this->get('doctrine.orm.entity_manager');  
$repository = $em->getRepository('<Bundle>\Entity\<Entite>');
```

Les requêtes avec DQL

Créer une requête

- DQL fournit une manière de créer des requêtes beaucoup plus proche de ce que l'on connaît avec PDO.

```
$query = $em->createQuery('SELECT u WHERE u.age > 20');  
$users = $query->getResult();
```

N.B. : La syntaxe SQL-like peut être trompeuse. On a bien à faire ici à des fonctions DQL

Les paramètres

- DQL supporte également les paramètres positionnés et nommés

```
$query = $em->createQuery(  
    'SELECT u FROM User u WHERE u.username = :name');  
$query->setParameter('name', 'Bob');  
$users = $query->getResult();
```

Les résultats

N.B. : En fonction des cas, DQL peut rendre une collection d'objets ou un tableau de valeurs (contrairement à QueryBuilder)

Différentes formes d'hydratation

- On peut choisir (partiellement) la forme sous laquelle les résultats sont présentés :

```
$query->getResult(); // dépend de la requête  
$query->getSingleResult(); // rend un objet  
$query->getArrayResult(); // rend les résultats sous forme de tableau  
$query->getScalarResult(); // rend un tableau « à plat »  
$query->getSingleScalarResult(); // dépend une valeur
```

Les Fonctions DANS DQL

Fonctions, opérateurs, agrégats

Les fonctions

- Comme QueryBuilder, DQL sait exprimer un sous-ensemble des fonctions SQL.
- La liste des fonctionnalités admises est en ligne dans la documentation de Doctrine

[Syntaxe abstraite de DQL](#)

Exemples

```
'SELECT u, COUNT(g.id) FROM User u JOIN u.groups g GROUP BY u.id');
```

```
'SELECT u.name FROM User u WHERE u.id IN(46, 53, 95, 144)';
```

```
'SELECT u.id FROM CmsUser u WHERE EXISTS (  
    SELECT p.phonenumber FROM CmsPhonenumber p WHERE p.user = u.id)  
'
```

PARTIAL

- Utiliser la fonction PARTIAL permet de récupérer des objets et non un tableau (hiérarchique) de valeurs

```
SELECT PARTIAL u.{id, username}, PARTIAL a.{id, name}
FROM User u
JOIN u.articles a
```

INDEX BY

- La fonction INDEX BY permet de choisir les valeurs des clefs du tableau associatif des résultats.

```
SELECT u, COUNT(g.id)
FROM User u
JOIN u.groups g GROUP BY u.id'
INDEX BY u.pseudo
```

Conclusion

A retenir

- DQL a une syntaxe proche de SQL
- DQL est la syntaxe sous-jacente à QueryBuilder
- Moins modulaire que QueryBuilder

Ce qu'on a vu

- La syntaxe de Doctrine Query Language
- Les différences avec QueryBuilder



Alphorm.com

Les modèles
Native SQL

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Présentation de Native SQL
- Le ResultSetMappingBuilder
- Les requêtes natives



Native SQL

Présentation

- Dans certains cas, on a besoin de concevoir des requêtes qui utilisent pleinement la syntaxe SQL
- On utilise alors Doctrine pour apparier les résultats sur les propriétés de certains objets/entités
- Il est nécessaire de déterminer cet appariement

Le resultSet
Déterminer l'appariement champs/propriétés

Le Result set

- des entités : les éléments racines des résultats
- des entités jointes : liées par des associations aux entités « racine »
- des champs : appartenant aux entités ci-dessus
- des valeurs scalaires :
- des méta-données : comme des clefs étrangères ou des colonnes de discrimination

Ajouter une entité

```
addJoinedEntityResult('Address' , 'a', 'u', 'address');
```

Ajouter une entité jointe

```
addJoinedEntityResult('User','u')
```

Ajouter un champ

```
addFieldResult('u', 'id_ville', 'idVille')
```

Ajouter une entité

```
addScalar('age','age')
```

Ajouter une entité

```
addMetaResult('u','adresse', 'adresse')
```

Ajouter une entité

```
$rsm->addMetaResult('u', 'discr', 'discr'); // discriminator column  
$rsm->setDiscriminatorColumn('u', 'discr');
```

Une requête native

Une requête native

- Créer une requête native se fait par la méthode `createNativeQuery`
- Le corps de la requête est du **SQL standard**

```
$query = $this->_em->createNativeQuery(
    'SELECT id, name FROM users WHERE name = ?',
    $rsm
);
$query->setParameter(1, 'romanb');
$users = $query->getResult();
```

Les résultats

N.B. 1 : En fonction de la requête, Doctrine rend des objets qui peuvent être partiels

N.B. 2 : Le ResultSetMapping est [persistant](#). Il faut donc l'annuler avec la fonction `clear()` une fois la requête achevée

Les requêtes nommées

Déclarer une requête nommée

- Un avantage de Native SQL est de pouvoir être déclarée dans le modèle de l'entité.
- On ajoute alors un élément `<sql-result-set-mappings>` pour déclarer les Result sets
- On ajoute également un élément `<named-native-queries>` pour les requêtes elles-mêmes

Exemple

```
<doctrine-mapping>
  <entity name="MyProject\Model\Address">
    <named-native-queries>
      <named-native-query name="findAll" result-set-mapping="mappingFindAll">
        <query>SELECT * FROM addresses</query>
      </named-native-query>
    </named-native-queries>
    <sql-result-set-mappings>
      <sql-result-set-mapping name="mappingFindAll">
        <entity-result entity-class="Address"/>
      </sql-result-set-mapping>
    </sql-result-set-mappings>
  </entity>
</doctrine-mapping>
```

Utilisation

- Une fois enregistrée, la requête nommée peut être utilisée directement dans le contrôleur

```
$em = $this->get('doctrine.entity_manager');
$rep = $em->getRepository(<Entite>);
$requete = $rep->createNamedQuery(<nomDeLaRequete>);
$requete->getResult();
```

Conclusion

A retenir

- Native SQL permet de créer des requêtes conformes à SQL, arbitrairement complexes
- La mise en œuvre est **plus lourde** que les autres méthodes
- Risque d'engendrer des **effets de bords**

Ce qu'on a vu

- Comment utiliser des requêtes SQL avec les entités Doctrine
- Comment paramétrer un Result set
- Quand utiliser cette syntaxe



Alphorm.com

Les services
Introduction

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES

Formateur et Consultant indépendant
Web, Gestion des connaissances

Plan

- Architecture de services
 - Définition des services
 - Créer un service
 - Déclarer un service à Symfony
 - Utiliser un service
- Un service simple



Qu'est-ce qu'un service ?

- Un service est un composant logiciel autonome réalisant une fonctionnalité clairement définie.
- Un service encapsule une collection d'opérations qui constituent autant d'actions spécifiques que le service peut effectuer.
- Un service coopère avec d'autres services disponibles pour répondre à une requête.

Caractéristiques d'un service

- Un **contrat** qui indique comment il peut être utilisé
- Un ensemble de dépendances faibles à d'autres services
- Une **abstraction** d'une partie de logique métier
- Une faculté à être **réutilisable**
- Une garantie **d'autonomie** (avec délégation éventuelle)
- Une possibilité d'être **découvert** (via un annuaire)
- Un élément essentiellement **composable**
- Une recherche de la **granularité optimale**

Architecture orientée services

- Montée en puissance de modèles SOA depuis quelques années
- Application au web avec les « **webservices** »
- Le web est une plate-forme très bien adaptée à ce type d'architecture
- Développement d'outils :
 - **Annuaire (UDDI)** pour la découverte de services
 - **WSDL, XML** pour la description des contrats
 - **SOAP** pour l'invocation des services
 - Utilisation de **HTTP** pour le transport

Les services dans Symfony

- Les services ne sont pas des « services web »
- Leur fonction n'est pas d'exposer un service sur le réseau
- Ils ont un rôle central dans Symfony comme outil de modularisation & composition du code
- Beaucoup d'objets d'une application peuvent être définis comme services et référencés comme tels dans l'annuaire nommé « conteneur de services »
- Les services sont des singletons (instanciés une seule fois)
- Les services sont hébergés par les bundles

Qu'est-ce qu'un service ?

- Un service est :
 1. une « vanilla PHP class ».
 2. une déclaration de service

Déclarer un service

- Déclarer un service est très simple, puisqu'il suffit d'exposer un fichier **services.*ml** dans le dossier :

Resources/config

- Il suffit juste de leur donner une étiquette et d'indiquer à Symfony la classe PHP correspondante :

```
<service id="tdn.image_processor" class="%image.processor.class%">
```

Services et contrôleurs

- Les objets héritant de la classe **Controller** (ou implémentant ContainerAware) ont un accès direct au contenu de services.
- Pour cette raison, invoquer un service à l'intérieur d'un contrôleur se fait par la méthode 'get'.

```
$processor = $this->get('tdn.image_processor');
```

Un service simple

Dans ce qui suit, nous allons développer un service simple
qui ira chercher des données sur des sites
externes.

Ce qu'on a vu

- Ce qu'était un service
- Comment sont utilisés les services dans Symfony
- Implémenter des services simples





Alphorm.com

Les services L'injection de dépendances

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation *Symfony3, les fondamentaux*

alphorm.com™ ©

Plan

- L'injection de dépendances (design pattern)
- Un service simple avec dépendances



Formation *Symfony3, les fondamentaux*

alphorm.com™ ©

Présentation

- L'injection de dépendances est un cadre de conception (*design pattern*) qui :

« consiste à créer dynamiquement (injecter) les dépendances entre les différentes classes en **s'appuyant sur une description** (fichier de configuration ou métadonnées) ou de manière programmatique.

Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais **déterminées dynamiquement à l'exécution.** »

[Wikipedia](#)

Trois types d'injections

- **Injection par constructeur** : les dépendances sont évaluées et créées lors de l'instanciation de l'objet de la classe de service,
- **Injection par méthode** : le service définit un mutateur (setter) qui est chargé de prendre en charge la dépendance,
- **Injection par interface** : on définit une interface qui permettra à tout service qui l'implémente de nouer une dépendance.

Les Arguments

Ajouter une dépendance : les arguments

- Dans Symfony, si l'on veut qu'un service dépende d'autres services, il faut l'indiquer dans la déclaration en ajoutant des arguments. Ceux-ci peuvent être de trois types :
 - d'autres services
 - des paramètres définis ailleurs dans l'application
 - des constantes
 - une expression

Impact sur la classe de service

- Les arguments doivent être insérés dans la signature du constructeur

=> il s'agit d'une **injection par constructeur**

```
<argument type='service'>App\Bundle\Service</argument>
```

Les « calls »

L'injection par mutateur

Dépendances de méthode : les calls

- Les calls sont un moyen d'ajouter de nouvelles dépendances, sans avoir à modifier le constructeur. Elles peuvent donc également devenir optionnelles. Les calls eux-mêmes font référence à un argument :

```
<call method="setMailer">  
  <argument type="service" id="app.mailer" />  
</call>
```

- Ici, il faudra de plus définir une méthode **semailles** dans la classe de service

Les Tags

Lier un service au noyau de Symfony

Qu'est-ce qu'un tag ?

- Les tags sont un moyen de déclarer au conteneur de services que la bibliothèque que vous publiez doit être prise en compte par tel ou tel composant (ou bundle tiers) d'une certaine manière
- Nous avons vu dans le chapitre consacré à Twig que, de ce cette manière, il était possible d'étendre le répertoire des filtres utilisables dans les squelettes de rendu.

Ajouter un tag

- Ajouter un tag un service, se fait de la même manière qu'un argument ou un call.

```
<tag name='twig.extension' />
```

- L'usage du service est de la responsabilité du composant auquel il est rattaché. Voir la documentation de Symfony pour les détails.

http://symfony.com/doc/current/reference/dic_tags.html

Exemples

- `form.type` : permet d'enregistrer de nouveaux types de champs formulaire
- `console.command` : ajouter une commande à la console de Symfony
- `kernel.event_listener` : ajouter une action à un hop du noyau
- `monolog.logger` : ajouter un canal de log à Monolog
- `routing.loader` : ajouter un méthode personnalisée de chargement des routes
- `templating.helper` : définir un service accessible dans les template PHP
- `validation.constraint_validator` : définir ses propres contraintes de validation

Un service avec dépendances

Implémentation

- Dans la démonstration qui suit, nous allons implémenter un service qui s'appuiera sur celui de la première partie (par injection) et proposera des méthodes pour post-traiter les informations reçues.

Ce qu'on a vu

- Le principe de l'injection de dépendances
- Les techniques dans Symfony
 - Déclarer un argument pour un service
 - Déclarer un call
- Comment associer un service à des composants du noyau de Symfony
- L'implémentation d'un service avec dépendances





Alphorm.com

Les services Tout est service

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Intégrer les services du noyau dans ses propres services
 - Les principaux services de Symfony
 - Accéder au routeur
- Les contrôleurs comme services
- Les formulaires comme services
- Personnaliser les fichiers de déclaration de services



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Les principaux services

- doctrine.orm.entity_manager
- event_dispatcher
- kernel
- logger
- mailer
- request_stack
- router
- security.context
- service_container
- twig
- templating

Exemple : accéder au routeur

```
<argument type='service' id='routeur' />
```

Accéder à l'environnement

- Les environnements sont principalement définis par leurs fichiers de configuration. C'est à priori la même base de code qui s'exécute.
- Néanmoins, on peut vouloir savoir à un moment du cycle dans quel environnement l'on se trouve.

```
/**
 * @return string le nom de l'environnement courant
 */
$env = $this->get('kernel')->getEnvironment();
```

Déclarer un contrôleur comme service

- Un contrôleur peut être un service, il suffit de le déclarer comme tel :

```
<service id="app.hello_controller" class="AppBundle\Controller\HelloController" />
```

- Il est ensuite possible de l'appeler :

```
$this->forward('app.hello_controller:indexAction', array('name' => $name));
```

Déclarer un contrôleur comme service

- Déclarer un contrôleur comme service rend aussi superflu la nécessité d'hériter de la classe Controller.
- Vous pouvez alors paramétrer exactement les services dont votre contrôleur aura besoin.

Déclarer un formulaire comme service

- De la même manière, il est possible de déclarer une classe de formulaire :

```
<service id="app.form.type.task" class='AppBundle\Form\Type\TaskType'>
  <tag name="form.type" alias='task_form' />
  <argument type="service" id=" app.my_service" />
</service>
```

Déclarer un formulaire comme service

- On peut ensuite directement utiliser l'alias dans le contrôleur :

```
$form = $this->createForm( 'task_form', $task );
```

Personnaliser les fichiers de déclaration

- Lorsque l'on crée un *bundle*, Symfony engendre automatiquement un dossier :

```
DependencyInjection
```

- Et à l'intérieur un fichier :

```
DependencyInjection/<nomDuBundle>Extension.php
```

Personnaliser les fichiers de déclaration

- La classe d'extension définit une méthode load qui permet de personnaliser l'emplacement des fichiers de déclaration de services

```
$loader->load('services.xml');
```

- En changeant le où du fichier (voire le répertoire) ou en ajoutant des lignes, il est possible de rendre plus modulaire la déclaration des services de l'application

Ce qu'on a vu

- Accéder à des services en dehors des contrôleurs
- Accéder aux configurations d'environnement
- Généraliser la notion de service
- Paramétrer la déclaration des services





Alphorm.com

Conclusion

Le mot de la fin

Site : <http://www.alphorm.com>
Blog : <http://blog.alphorm.com>



Michel CADENNES
Formateur et Consultant indépendant
Web, Gestion des connaissances

Formation Symfony3, les fondamentaux

alphorm.com™ ©

Plan

- Rappel des objectifs du cours
- Une synthèse des points abordés dans les différents chapitres
- Ce qui n'a pas été couvert, mais le sera par la suite



Formation Symfony3, les fondamentaux

alphorm.com™ ©

Objectifs

- Le but de ce cours était d'expliquer les fondements de l'architecture d'une application dans l'environnement Symfony
 - Découvrir **les composants principaux** proposés par Symfony
 - Comprendre **le cycle d'exécution** d'une requête
 - **Créer et organiser son code** pour construire une application simple
 - Repérer **les bonnes pratiques** de Symfony et de PHP en général
 - Intégrer **les fonctionnalités avancées de PHP 5.3+** et les contraintes de Symfony

Une petite synthèse

PHP

- Depuis **PHP 5.3** de nombreuses fonctionnalités ont été ajoutées :
 - espaces de noms (5.3)
 - import de fonctions et de constantes séparées (5.6)
 - fonctions anonymes (5.3)
 - traits (5.4)
 - résolution statique à la volée (5.3)
 - opérateur de décomposition (5.6)
- PHP offre de nouvelles possibilités d'adopter un style de programmation fonctionnelle :
 - générateurs (5.5)
 - découverte de type généralisée (7.0)

Composer

- Le gestionnaire de paquets pour PHP
- Automatise la mise à jour et la migration d'applications
- Repose sur le dépôt **Packagist.com**

Installation & configuration

- Symfony propose plusieurs méthodes pour le « **bootstrap** » des applications
- Les langages et formats pour la configuration des applications sont très variés
- D'une manière générale, Symfony laisse une grande marge de manœuvre aux développeurs
- Il est fortement conseillé, néanmoins, de suivre les principales recommandations PSR-x

Les bundles

- Un bundle (ou « bouquet » en français) est une **unité fonctionnelle** qui agence les ressources (MVC, notamment) que Symfony attend
- Les bundles doivent être pensés comme étant **réutilisables** dans d'autres contextes (et potentiellement distribuables)
- Les bundles peuvent se constituer en **hiérarchie** et **hériter** les uns des autres

Routage

- Les routes se définissent par l'intermédiaire de **fichiers de configuration**
- Ces fichiers sont associés aux bundles, mais peuvent être **distribués** ou **globalisés**
- Le routeur discrimine les routes en fonction de **contraintes déclarées**
- Le routeur peut être appelé en tant que **service**

Les contrôleurs

- Le contrôleur est la « **tour de contrôle** » du cycle d'exécution
- Il a un accès direct au conteneur de services via l'interface **ContainerAwareInterface**
- Son rôle doit se limiter à :
 - La réception de la requête HTTP
 - L'appel à des services pour le traitement des données
 - ... en particulier la délégation de la gestion de la persistance du modèle
 - L'envoi de la réponse HTTP

Les vues

- Les vues peuvent être gérées de deux manières :
 - en PHP natif
 - en recourant au moteur de template Twig
- Twig offre la possibilité de :
 - rendre les squelettes des vues modulaires
 - disposer d'une abstraction, davantage portable
 - concevoir facilement des extensions de la syntaxe de base
- D'autres syntaxes (Smarty, Markdown) peuvent être facilement intégrées à Symfony

Les formulaires

- Les formulaires sont définies de manière déclarative dans des classes spécifiques
- Le rendu des formulaires de fait fait par la configuration de champs de formulaires, eux-mêmes extensibles
- Le traitement des formulaires est pris en charge par Doctrine via les contrôleurs
- Symfony offre un système évolué de validation pour les données des formulaires

Les modèles

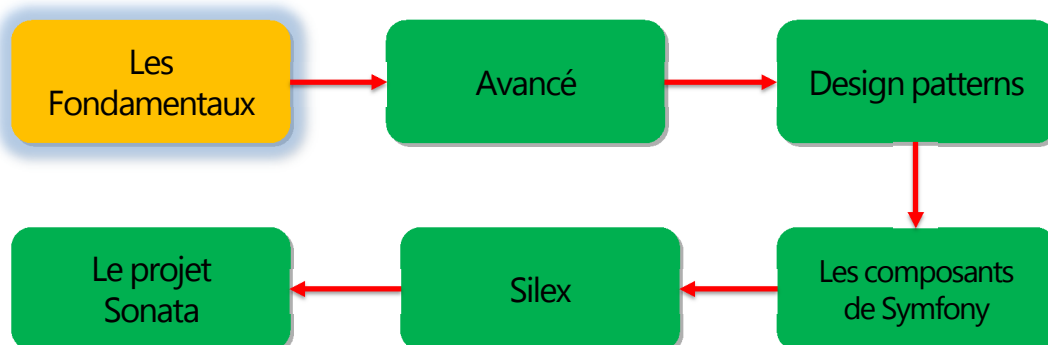
- Par défaut, Symfony utilise l'ORM Doctrine comme outil de persistance des modèles
- Doctrine prend en charge la chaîne de réification du modèle conceptuel des données
- En tant qu'ORM, Doctrine offre une abstraction objet du schéma SQL associé à l'application
- Doctrine autorise la construction de requêtes dans plusieurs langages de plus ou moins « haut niveau »

Les services

- La notion de service est la pierre angulaire de la modularisation du code dans Symfony
- Un service n'est en soi qu'une classe vanilla-PHP
 - Toute classe peut devenir un service dans Symfony
- Les services n'ont d'utilité que par la combinaison de :
 - L'agrégation des services au sein du conteneur de services
 - L'assemblage des services par biais de l'injection de dépendances
- Des services peuvent être collectés pour étendre les fonctionnalités d'autres services

Encore à venir

 Cusrus Symfony



Ce qui reste à voir

- La gestion des utilisateurs
- La sécurité
- Les événements
- Le fonctionnement de la console
- Les applications multilingues
- Les différents caches
- La configuration de services
- Les politiques de tests (PHPUnit)
- La création de services web
- Les SGBD non SQL
- etc.