



Développez votre site web avec le framework Symfony2

Par Alexandre Bacco ([winzou](#))



Sommaire

Sommaire	2
Partager	6
Développez votre site web avec le framework Symfony2	8
Partie 1 : Vue d'ensemble de Symfony2	9
Symfony2, un framework PHP	9
Qu'est-ce qu'un framework ?	9
L'objectif d'un framework	9
Pesons le pour et le contre	9
Alors, convaincus ?	10
Qu'est-ce que Symfony2 ?	10
Un framework	10
Un framework populaire	10
Un framework populaire et français	11
Télécharger Symfony2	11
Obtenir Symfony2	11
Vérifier votre configuration de PHP	11
Vérifier l'installation de PHP en console	13
En résumé	14
Vous avez dit Symfony2 ?	15
L'architecture des fichiers	15
Liste des répertoires	15
Le contrôleur frontal	16
L'architecture conceptuelle	17
Architecture MVC	17
Parcours d'une requête dans Symfony2	17
Symfony2 et ses bundles	19
La découpe en bundles	19
La structure d'un bundle	20
En résumé	20
Utilisons la console pour créer un bundle	21
Utilisation de la console	22
Sous Windows	22
Sous Linux et Mac	23
À quoi ça sert ?	23
Comment ça marche ?	23
Créons notre bundle	24
Tout est bundle	24
Que s'est-il passé ?	27
À retenir	29
En résumé	29
Partie 2 : Les bases de Symfony2	30
Mon premier « Hello World ! » avec Symfony2	31
Créons notre route	31
Le routeur (ou router) ? Une route ?	31
Créons notre contrôleur	33
Le rôle du contrôleur	33
Créons notre contrôleur	34
Créons notre template Twig	35
Les templates avec Twig	35
Utiliser Twig avec Symfony2	36
Notre objectif : créer un blog	38
Le fil conducteur : un blog	38
Notre blog	38
Un peu de nettoyage	38
Schéma de développement sous Symfony2	38
Pour conclure	39
En résumé	39
Le routeur de Symfony2	40
Le fonctionnement	41
Fonctionnement du routeur	41
Convention pour le nom du contrôleur	43
Les routes de base	43
Créer une route	43
Créer une route avec des paramètres	44
Les routes avancées	46
Créer une route avec des paramètres et leurs contraintes	46
Utiliser des paramètres facultatifs	47
Utiliser des « paramètres système »	47
Ajouter un préfixe lors de l'import de nos routes	47
Générer des URL	48
Pourquoi générer des URL ?	48
Comment générer des URL ?	48
Application : les routes de notre blog	49
Construction des routes	49

Récapitulatif	50
Pour conclure	51
En résumé	51
Les contrôleurs avec Symfony2	51
Le rôle du contrôleur	52
Retourner une réponse	52
Manipuler l'objet Request	52
Les paramètres de la requête	52
Les autres méthodes de l'objet Request	54
Manipuler l'objet Response	55
Décomposition de la construction d'un objet Response	55
Réponses et vues	56
Réponse et redirection	57
Changer le Content-type de la réponse	57
Les différents services	58
Qu'est-ce qu'un service ?	58
Accéder aux services	58
Brève liste des services	59
Application : le contrôleur de notre blog	63
Construction du contrôleur	63
À retenir	65
Testons-le	65
Pour conclure	66
En résumé	66
Le moteur de templates Twig	66
Les templates Twig	67
Intérêt	67
En pratique	67
À savoir	68
Afficher des variables	68
Syntaxe de base pour afficher des variables	68
Précisions sur la syntaxe {{ objet.attribut }}	69
Les filtres utiles	69
Twig et la sécurité	70
Les variables globales	70
Structures de contrôle et expressions	71
Les structures de contrôle	71
Les tests utiles	73
Hériter et inclure des templates	74
L'héritage de template	74
Qu'est-ce que l'on vient de faire ?	75
Le modèle « triple héritage »	75
L'inclusion de templates	76
L'inclusion de contrôleurs	78
Application : les templates de notre blog	79
Layout général	80
Layout du bundle	81
Les templates finaux	82
Pour conclure	89
En résumé	89
Installer un bundle grâce à Composer	89
Composer, qu'est-ce que c'est ?	90
Un gestionnaire de dépendances	90
Comment Composer sait où trouver les bibliothèques ?	90
Un outil innovant... dans l'écosystème PHP	90
Concrètement, comment ça marche ?	90
Installer Composer et Git	90
Installer Composer	90
Installer Git	91
Installer un bundle grâce à Composer	92
Manipulons Composer	92
Mettons à jour Symfony2	93
Installer un bundle avec Composer	94
Gérer l'autoload d'une bibliothèque manuellement	96
Pour conclure	97
En résumé	97
Les services, théorie et création	98
Pourquoi utiliser des services ?	98
Genèse	98
Qu'est-ce qu'un service ?	98
Le conteneur de services	98
Comment définir les dépendances entre services ?	100
La persistance des services	100
Utiliser un service en pratique	100
Récupérer un service	100
Créer un service simple	101
Création de la classe du service	101
Création de la configuration du service	102
Utilisation du service	104
Créer un service avec des arguments	105
Passer à la vitesse supérieure	105
Injecter des arguments dans nos services	105
L'injection de dépendances	106

Pour conclure	107
En résumé	107
Partie 3 : Gérer la base de données avec Doctrine2	107
La couche métier : les entités	108
Notions d'ORM : fini les requêtes, utilisons des objets	108
Vos données sont des objets	108
Créer une première entité avec Doctrine2	108
Une entité, c'est juste un objet	108
Une entité, c'est juste un objet... mais avec des commentaires !	110
Générer une entité : le générateur à la rescousse !	111
Affiner notre entité avec de la logique métier	114
Conclusion	115
Tout sur le mapping !	115
L'annotation Entity	115
L'annotation Table	116
L'annotation Column	116
Pour conclure	118
En résumé	118
Manipuler ses entités avec Doctrine2	118
Matérialiser les tables en base de données	119
Créer la table correspondante dans la base de données	119
Modifier une entité	120
À retenir	122
Enregistrer ses entités avec l'EntityManager	122
Les services Doctrine2	122
Enregistrer ses entités en base de données	124
Récupérer ses entités avec un EntityRepository	128
En résumé	130
Les relations entre entités avec Doctrine2	130
Présentation	131
Présentation	131
Les différents types de relations	131
Notions techniques d'ORM à savoir	131
Rien n'est magique	132
Relation One-To-One	132
Présentation	132
Définition de la relation dans les entités	133
Exemple d'utilisation	136
Relation Many-To-One	138
Présentation	138
Définition de la relation dans les entités	141
Exemple d'utilisation	143
Relation Many-To-Many	144
Présentation	144
Définition de la relation dans les entités	146
Remplissons la base de données avec les fixtures	149
Exemples d'utilisation	150
Relation Many-To-Many avec attributs	152
Présentation	152
Définition de la relation dans les entités	154
Remplissons la base de données	156
Exemple d'utilisation	157
Les relations bidirectionnelles	160
Présentation	160
Définition de la relation dans les entités	161
Pour conclure	164
En résumé	164
Récupérer ses entités avec Doctrine2	166
Le rôle des repositories	166
Définition	166
Les méthodes de récupération des entités	166
Les méthodes de récupération de base	167
Définition	167
Les méthodes normales	167
Les méthodes magiques	169
Les méthodes de récupération personnelles	169
La théorie	169
Utilisation du Doctrine Query Language (DQL)	176
Utiliser les jointures dans nos requêtes	177
Pourquoi utiliser les jointures ?	177
Comment faire des jointures avec le QueryBuilder ?	178
Comment utiliser les jointures ?	179
Application : les entités de notre blog	179
Plan d'attaque	179
À vous de jouer !	179
Le code	180
En résumé	180
Les évènements et extensions Doctrine	181
Les évènements Doctrine	181
L'intérêt des évènements Doctrine	181
Définir des callbacks de cycle de vie	181
Liste des évènements de cycle de vie	182

Un autre exemple d'utilisation	183
Les extensions Doctrine	184
L'intérêt des extensions Doctrine	184
Installer le StofDoctrineExtensionBundle	185
Utiliser une extension : l'exemple de Sluggable	185
Liste des extensions Doctrine	187
Pour conclure	187
En résumé	187
TP : Les entités de notre blog	188
Synthèse des entités	189
Entité Article	189
Entité Image	193
Entité Commentaire	195
Entité Categorie	197
Entités Competence et ArticleCompetence	198
Et bien sûr...	200
Adaptation du contrôleur	200
Théorie	200
Pratique	201
Amélioration du contrôleur	203
L'utilisation d'un ParamConverter	204
Utiliser une jointure pour récupérer les articles	205
La pagination des articles sur la page d'accueil	206
Améliorer les vues	209
Pour conclure	211
En résumé	211
Partie 4 : Allons plus loin avec Symfony2	211
Créer des formulaires avec Symfony2	212
Gestion des formulaires	212
L'enjeu des formulaires	212
Un formulaire Symfony2, qu'est-ce que c'est ?	212
Gestion basique d'un formulaire	213
Gestion de la soumission d'un formulaire	217
Gérer les valeurs par défaut du formulaire	219
Personnaliser l'affichage d'un formulaire	220
Créer des types de champ personnalisés	221
Externaliser la définition de ses formulaires	221
Définition du formulaire dans ArticleType	221
Le contrôleur épuré	222
Les formulaires imbriqués	223
Intérêt de l'imbrication	223
Un formulaire est un champ	224
Relation simple : imbriquer un seul formulaire	225
Relation multiple : imbriquer un même formulaire plusieurs fois	227
Un type de champ très utile : entity	231
Aller plus loin avec les formulaires	234
L'héritage de formulaire	234
Construire un formulaire différemment selon des paramètres	235
Le type de champ File pour envoyer des fichiers	237
Le type de champ File	237
Automatiser le traitement grâce aux évènements	240
Application : les formulaires de notre blog	244
Théorie	244
Pratique	244
Pour conclure	250
En résumé	250
Validez vos données	250
Pourquoi valider des données ?	251
Never trust user input	251
L'intérêt de la validation	251
La théorie de la validation	251
Définir les règles de validation	251
Les différentes formes de règles	251
Définir les règles de validation	251
Déclencher la validation	256
Le service Validator	256
La validation automatique sur les formulaires	257
Conclusion	257
Encore plus de règles de validation	257
Valider depuis un getter	257
Valider intelligemment un attribut objet	258
Valider depuis un Callback	259
Valider un champ unique	260
Valider selon nos propres contraintes	261
1. Créer la contrainte	261
2. Créer le validateur	261
3. Transformer son validateur en service	263
Pour conclure	265
En résumé	265
Sécurité et gestion des utilisateurs	265
Authentification et autorisation	266
Les notions d'authentification et d'autorisation	266

Exemples	266
Processus général	270
Première approche de la sécurité	270
Le fichier de configuration de la sécurité	271
Mettre en place un pare-feu	273
Les erreurs courantes	279
Récupérer l'utilisateur courant	280
Gestion des autorisations avec les rôles	281
Définition des rôles	281
Tester les rôles de l'utilisateur	282
Utiliser des utilisateurs de la base de données	285
Qui sont les utilisateurs ?	285
Créons notre classe d'utilisateurs	285
Créons quelques utilisateurs de test	287
Définissons l'encodeur pour notre nouvelle classe d'utilisateurs	288
Définissons le fournisseur d'utilisateurs	288
Manipuler vos utilisateurs	290
Utiliser FOSUserBundle	290
Installation de FOSUserBundle	290
Configuration de la sécurité pour utiliser le bundle	293
Configuration du bundle FOSUserBundle	295
Personnalisation esthétique du bundle	296
Manipuler les utilisateurs avec FOSUserBundle	298
Pour conclure	299
Les services, utilisation poussée	299
Les tags sur les services	300
Les tags ?	300
Comprendre les tags à travers Twig	300
Les principaux tags	302
Dépendances optionnelles : les calls	304
Les dépendances optionnelles	304
Les calls	305
L'utilité des calls	305
Les champs d'application, ou scopes	305
La problématique	305
Que sont les scopes ?	306
Et concrètement ?	306
Les services courants de Symfony2	307
Les services courants de Symfony	307
En résumé	308
Le gestionnaire d'événements de Symfony2	308
Des événements ? Pour quoi faire ?	309
Qu'est-ce qu'un événement ?	309
Qu'est-ce que le gestionnaire d'événements ?	309
Écouter les événements	310
Notre exemple	310
Créer un listener	310
Écouter un événement	311
Méthodologie	316
Les événements Symfony2... et les nôtres !	316
Les événements Symfony2	316
Créer nos propres événements	320
Allons un peu plus loin	325
Les souscripteurs d'événements	325
L'ordre d'exécution des listeners	326
La propagation des événements	327
En résumé	327
Traduire son site	327
Introduction à la traduction	328
Le principe	328
Prérequis	329
Bonjour le monde	330
Dire à Symfony « Traduis-moi cela »	330
Notre vue	332
Le catalogue	333
Les formats de catalogue	333
Notre traduction	334
Récupérer la locale de l'utilisateur	337
Déterminer la locale	337
Routing et locale	337
Les paramètres par défaut	339
Organiser vos catalogues	339
Utiliser des mots-clés plutôt que du texte comme chaînes sources	339
Permettre le retour à la ligne au milieu des chaînes cibles	341
Utiliser des listes	342
Utiliser les domaines	343
Traductions dépendantes de variables	344
Les placeholders	344
Gestion des pluriels	346
Afficher des dates au format local	348
Pour conclure	349
En résumé	350

Partie 5 : Astuces et points particuliers	350
Utiliser des ParamConverters pour convertir les paramètres de requêtes	351
Théorie : pourquoi un ParamConverter ?	351
Récupérer des entités Doctrine avant même le contrôleur	351
Les ParamConverters	351
Un ParamConverter utile : DoctrineParamConverter	351
Un peu de théorie sur les ParamConverters	352
Pratique : utilisation des ParamConverters existants	352
Utiliser le ParamConverter Doctrine	352
Utiliser le ParamConverter Datetime	355
Aller plus loin : créer ses propres ParamConverters	356
Comment sont exécutés les ParamConverters ?	356
Comment Symfony2 trouve tous les convertisseurs ?	357
Créer un convertisseur	357
L'exemple de notre TestParamConverter	358
Personnaliser les pages d'erreur	360
Théorie : remplacer les vues d'un bundle	361
Constateter les pages d'erreur	361
Localiser les vues concernées	361
Remplacer les vues d'un bundle	361
Comportement de Twig	362
Pourquoi il y a tous les formats error.XXX.twig dans le répertoire Exception ?	362
Pratique : remplacer les templates Exception de TwigBundle	362
Créer la nouvelle vue	362
Le contenu d'une page d'erreur	363
En résumé	363
Utiliser Assetic pour gérer les codes CSS et JS de votre site	363
Théorie : entre vitesse et lisibilité, pourquoi choisir ?	364
À propos du nombre de requêtes HTTP d'une page web	364
Comment optimiser le front-end ?	364
Il est aussi possible d'améliorer le temps de chargement !	364
Conclusion	364
Pratique : Assetic à la rescousse !	365
Servir des ressources	365
Modifier les ressources servies	367
Gestion du mode prod	369
Et bien plus encore...	370
Utiliser la console directement depuis le navigateur	371
Théorie : le composant Console de Symfony2	371
Les commandes sont en PHP	371
Exemple d'une commande	371
Pratique : utiliser un ConsoleBundle	373
ConsoleBundle ?	373
Installer CoreSphereConsoleBundle	373
Utilisation de la console dans son navigateur	376
Prêts pour l'hébergement mutualisé	376
En résumé	376
Déployer son site Symfony2 en production	377
Préparer son application en local	377
Vider le cache, tout le cache	377
Tester l'environnement de production	377
Soigner ses pages d'erreur	378
Installer une console sur navigateur	378
Vérifier et préparer le serveur de production	378
Vérifier la compatibilité du serveur	378
Modifier les paramètres OVH pour être compatible	380
Déployer votre application	380
Envoyer les fichiers sur le serveur	380
Régler les droits sur les dossiers app/cache et app/logs	380
S'autoriser l'environnement de développement	381
Mettre en place la base de données	381
S'assurer que tout fonctionne	382
Avoir de belles URL	382
Et profitez !	383
Les outils pour déployer votre projet	383
En résumé	383
Toutes les bonnes choses ont une fin !	384
Le code complet du cours	384
Plus de lecture sur mon blog	384
Licences	384



Symfony

Développez votre site web avec le framework Symfony2



Le tutoriel que vous êtes en train de lire est en **bêta-test**. Son auteur souhaite que vous lui fassiez part de vos commentaires pour l'aider à l'améliorer avant sa publication officielle. Notez que le contenu n'a pas été validé par l'équipe éditoriale du Site du Zéro.

Par



Alexandre Bacco (winzou)

Mise à jour : 03/05/2013

Difficulté : Intermédiaire Durée d'étude : 1 mois



20 253 visites depuis 7 jours, classé 15/806

Vous savez déjà faire des sites internet ? Vous maîtrisez votre code, mais n'êtes pas totalement satisfait ? Vous avez trop souvent l'impression de réinventer la roue ?

Alors ce tutoriel est fait pour vous !

Symfony2 est un puissant framework qui va vous permettre de réaliser des sites complexes rapidement, mais de façon structurée et avec un code clair et maintenable. En un mot : le paradis du développeur !

Ce tutoriel est un tutoriel pour débutants sur Symfony2, vous n'avez besoin d'aucune notion sur les *frameworks* pour l'aborder, nous allons les découvrir ensemble. Cependant, il est fortement conseillé :

- D'avoir déjà une bonne expérience de PHP ([aller au cours Concevez votre site web avec PHP et MySQL](#)) ;
- De maîtriser les notions de base de la POO ([aller au cours La programmation orientée objet](#)) ;
- D'avoir éventuellement des notions de namespace ([aller au cours Les espaces de nom](#)).



Si vous ne maîtrisez pas ces trois points, je vous invite vraiment à les apprendre avant de commencer la lecture de ce cours. Symfony2 requiert ces bases, et, si vous ne les avez pas, vous risquez de mettre plus de temps pour assimiler ce tutoriel. C'est comme acheter un A380 sans savoir piloter : c'est joli, mais cela ne sert à rien.

Partie 1 : Vue d'ensemble de Symfony2

Pour débuter, quoi de mieux que de commencer par le commencement ! Si vous n'avez aucune expérience dans les *frameworks* ni dans l'architecture MVC, cette partie sera très riche en nouvelles notions. Avançons doucement mais sûrement, vous êtes là pour apprendre !

Symfony2, un framework PHP

Alors, vous avez décidé de vous lancer dans Symfony2 ? Parfait, vous ne le regretterez pas ! Tout au long de ce cours, nous apprendrons à utiliser ce framework, et vous comprendrez petit à petit la puissance de cet outil. Mais tout d'abord, commençons par les bases, et voyons précisément quels sont les objectifs et les limites d'un framework tel que Symfony2.

Dans ce chapitre, nous allons découvrir pourquoi Symfony2 est un bon choix pour votre application web. Une boîte à outils faite en PHP qui a pour but de vous simplifier la vie, c'est toujours sympa, non ? Allons-y !

Qu'est-ce qu'un framework ?

L'objectif d'un framework

L'objectif de ce chapitre n'est pas de vous fournir toutes les clés pour *concevoir* un framework, mais suffisamment pour pouvoir en *utiliser* un. On exposera rapidement l'intérêt, les avantages et les inconvénients de l'utilisation d'un tel outil.

Définition

Le mot « *framework* » provient de l'anglais « *frame* » qui veut dire « cadre » en français, et « *work* » qui signifie « travail ». Littéralement, c'est donc un « cadre de travail ». Vous voilà avancés, hein ? 😊 Concrètement, c'est un ensemble de composants qui servent à créer les fondations, l'architecture et les grandes lignes d'un logiciel. Il existe des centaines de frameworks couvrant la plupart des langages de programmation. Ils sont destinés au développement de sites web ou bien à la conception de logiciels.

Un framework est une boîte à outils conçue par un ou plusieurs développeurs à destination d'autres développeurs. Contrairement à certains scripts tels que WordPress, Dotclear ou autres, un framework n'est pas utilisable tel quel. Il n'est pas fait pour être utilisé par les utilisateurs finaux. Le développeur qui se sert d'un framework a encore du boulot à fournir, d'où ce cours !

Objectif d'un framework

L'objectif premier d'un framework est d'améliorer la productivité des développeurs qui l'utilisent. Plutôt sympa, non ? Souvent organisé en différents composants, un framework offre la possibilité au développeur final d'utiliser tel ou tel composant pour lui faciliter le développement, et lui permet ainsi de se concentrer sur le plus important.

Prenons un exemple concret. Il existe dans Symfony2 un composant qui gère les formulaires HTML : leur affichage, leur validation, etc. Le développeur qui l'utilise se concentre sur l'essentiel dans son application : chaque formulaire effectue une action, et c'est cette action qui est importante, pas les formulaires. Étendez ce principe à toute une application ou tout un site internet, et vous comprenez l'intérêt d'un framework ! Autrement dit, le framework s'occupe de la forme et permet au développeur de se concentrer sur le fond.

Pesons le pour et le contre

Comme tout bon développeur, lorsqu'on veut utiliser un nouvel outil, on doit en peser le pour et le contre pour être sûr de faire le bon choix !

Les pour

L'avantage premier est donc, on vient de le voir, le gain en productivité. Mais il en existe bien d'autres ! On peut les classer en plusieurs catégories : le code, le travail et la communauté.

Tout d'abord, un framework va vous aider à réaliser un « bon code ». Par « bon code », j'entends qu'il vous incite, de par sa propre architecture, à bien organiser votre code. Et un code bien organisé est un code facilement maintenable et évolutif ! De plus, un framework offre des briques prêtées à être utilisées (le composant *Formulaire* de Symfony2 par exemple), ce qui vous évite de réinventer la roue, et surtout qui vous permet d'utiliser des briques puissantes et éprouvées. En effet, ces briques sont développées par des équipes de développeurs chevronnés, elles sont donc très flexibles et très robustes. Vous économisez ainsi

des heures de développement !

Ensuite, un framework améliore la façon dont vous travaillez. En effet, dans le cas d'un site internet, vous travaillez souvent avec d'autres développeurs PHP et un designer. Un framework vous aide doublement dans ce travail en équipe. D'une part, un framework utilise presque toujours l'architecture MVC ; on en reparlera, mais sachez pour le moment que c'est une façon d'organiser son code qui sépare le code PHP du code HTML. Ainsi, votre designer peut travailler sur des fichiers différents des vôtres, fini les problèmes d'édition simultanée d'un même fichier ! D'autre part, un framework a une structure et des conventions de code connues. Ainsi, vous pouvez facilement recruter un autre développeur : s'il connaît déjà le framework en question, il s'intégrera très rapidement au projet.

Enfin, le dernier avantage est la communauté soutenant chaque framework. C'est elle qui fournit les tutoriaux ou les cours (comme celui que vous lisez !), de l'aide sur les forums, et bien sûr les mises à jour du framework. Ces mises à jour sont très importantes : imaginez que vous codiez vous-mêmes tout ce qui est connexion utilisateur, session, moteur de *templates*, etc. Comme il est impossible de coder sans bugs, vous devriez logiquement corriger chaque bug déclaré sur votre code. Maintenant, imaginez que toutes les briques de votre site, qui ne sont pas forcément votre tasse de thé, soient fournies par le framework. À chaque fois que vous ou les milliers d'autres utilisateurs du framework trouverez un bug, les développeurs et la communauté s'occuperont de le corriger, et vous n'aurez plus qu'à suivre les mises à jour. Un vrai paradis !

Il existe plein d'autres avantages que je ne vais pas vous détailler, mais un framework, c'est aussi :

- Une communauté active qui utilise le framework et qui contribue en retour ;
- Une documentation de qualité et régulièrement mise à jour ;
- Un code source maintenu par des développeurs attitrés ;
- Un code qui respecte les standards de programmation ;
- Un support à long terme garanti et des mises à jour qui ne cassent pas la compatibilité ;
- Etc.

Les contre

Vous vous en doutez, avec autant d'avantages il y a forcément des inconvénients. Eh bien, figurez-vous qu'il n'y en a pas tant que ça !

S'il ne fallait en citer qu'un, cela serait évidemment la courbe d'apprentissage qui est plus élevée. En effet, pour maîtriser un framework, il faut un temps d'apprentissage non négligeable. Chaque brique qui compose un framework a sa complexité propre qu'il vous faudra appréhender.

Notez également que pour les frameworks les plus récents, tels que Symfony2 justement, il faut également être au courant des dernières nouveautés de PHP. Je pense notamment à la programmation orientée objet et aux namespaces. De plus, connaître certaines bonnes pratiques telles que l'architecture MVC est un plus.

Mais rien de tout cela ne doit vous effrayer ! Voyez l'apprentissage d'un framework comme un investissement : il y a un certain effort à fournir au début, mais les résultats se récoltent ensuite sur le long terme !

Alors, convaincus ?

J'espère vous avoir convaincus que les pour l'emportent largement sur les contre. Si vous êtes prêts à relever le défi aujourd'hui pour être plus productifs demain, alors ce cours est fait pour vous !

Qu'est-ce que Symfony2 ?

Un framework

Symfony2 est donc un framework PHP. Bien sûr, il en existe d'autres ; pour ne citer que les plus connus : Zend Framework, CodeIgniter, CakePHP, etc. Le choix d'un framework est assez personnel, et doit être adapté au projet engagé. Sans vouloir prêcher pour ma paroisse, Symfony2 est l'un des plus flexibles et des plus puissants.

Un framework populaire

Symfony est très populaire. C'est un des frameworks les plus utilisés dans le monde, notamment dans les entreprises. Il est utilisé par Dailymotion par exemple ! La première version de Symfony est sortie en 2005 et est aujourd'hui toujours très utilisée. Cela lui apporte un retour d'expérience et une notoriété exceptionnels. Aujourd'hui, beaucoup d'entreprises dans le domaine de l'internet

(dont Simple IT, l'éditeur du Site du Zéro !) recrutent des développeurs capables de travailler sous ce framework. Ces développeurs pourront ainsi se greffer aux projets de l'entreprise très rapidement, car ils en connaîtront déjà les grandes lignes. C'est un atout si vous souhaitez travailler dans ce domaine. 😊

La deuxième version, que nous étudierons dans ce tutoriel, est sortie en août 2011. Elle est encore jeune, son développement a été fulgurant grâce à une communauté de développeurs dévoués. Bien que différente dans sa conception, cette deuxième version est plus rapide et plus souple que la première. Il y a fort à parier que très rapidement beaucoup d'entreprises s'arracheront les compétences des premiers développeurs Symfony2. Faites-en partie !

Un framework populaire et français

Eh oui, Symfony2, l'un des meilleurs frameworks PHP au monde, est un framework français ! Il est édité par la société [SensioLabs](#), dont le créateur est Fabien Potencier. Mais Symfony2 étant un script *open source*, il a également été écrit par toute la communauté : beaucoup de Français, mais aussi des développeurs de tous horizons : Europe, États-Unis, etc. C'est grâce au talent de Fabien et à la générosité de la communauté que Symfony2 a vu le jour.

Télécharger Symfony2

Obtenir Symfony2

Il existe de nombreux moyens d'obtenir Symfony2. Nous allons voir ici la méthode la plus simple : télécharger la distribution standard. Pour cela, rien de plus simple, rendez-vous sur le site de Symfony2, rubrique [Download](#), et téléchargez la version « Symfony Standard (.zip) ».



Si vous utilisez déjà Composer, vous pouvez télécharger cette distribution standard en une seule commande :

```
php composer.phar create-project symfony/framework-standard-edition Symfony 2.2
```

Cela va installer la version 2.2 de Symfony2 dans le répertoire Symfony.



Si vous êtes sous Windows, évitez de télécharger l'archive au format .tgz car des problèmes ont été rencontrés avec cette archive.



Ce cours a été écrit pour la version 2.2 de Symfony. Cependant, une attention particulière a été apportée pour qu'il soit compatible avec la version 2.3 au minimum. Vous ne devriez donc pas avoir de soucis à l'utiliser sur les versions 2.3 et 2.4, mais gardez ce point en tête si jamais.

Une fois l'archive téléchargée, décompressez les fichiers dans votre répertoire web habituel, par exemple C :\wamp\www pour Windows ou /var/www pour Linux. Pour la suite du tutoriel, je considérerai que les fichiers sont accessibles à l'URL <http://localhost/Symfony>. Je vous recommande d'avoir la même adresse, car je ferai ce genre de liens tout au long du tutoriel. 😊

Vérifier votre configuration de PHP

Symfony2 a quelques contraintes par rapport à votre configuration PHP. Par exemple, il ne tourne que sur la version 5.3.2 ou supérieure de PHP. Pour vérifier si votre environnement est compatible, rendez-vous à l'adresse suivante : <http://localhost/Symfony/web/config.php>. Si vous avez une version adéquate de PHP, vous devriez obtenir la figure suivante.

The screenshot shows the Symfony2 welcome page. It features the Symfony logo (a stylized 'sf' inside a circle) and the word "Symfony". The main heading is "Welcome!" followed by the subtext "Welcome to your new Symfony project." Below this, it says "This script will guide you through the basic configuration of your project. You can also do the same by editing the 'app/config/parameters.ini' file directly." A green box labeled "RECOMMENDATIONS" contains three points: 1. Install and enable a PHP accelerator like APC (highly recommended). 2. Upgrade your intl extension with a newer ICU version (4+). 3. Set short_open_tag to off in php.ini*. At the bottom, there are three links: "Configure your Symfony Application online >", "Bypass configuration and go to the Welcome page >", and "Re-check configuration >".

Mon environnement de travail est compatible avec Symfony2 !

En cas d'incompatibilité (version de PHP notamment), Symfony2 vous demande de régler les problèmes avant de continuer. Si l'on ne vous propose que des recommandations, vous pouvez continuer sans problème. Ce sont des points que je vous conseille de régler, mais qui sont facultatifs.

Je fais un petit aparté pour les lecteurs travaillant sous Linux. Symfony2 a besoin d'écrire dans quelques répertoires, il faut donc bien régler les droits sur les répertoires app/cache et app/logs. Pour cela, placez-vous dans le répertoire Symfony et videz d'abord ces répertoires :

Code : Console

```
rm -rf app/cache/*
rm -rf app/logs/*
```

Ensuite, si votre distribution supporte le chmod +a, exécutez ces commandes pour définir les bons droits :

Code : Console

```
sudo chmod +a "www-data allow delete,write,append,file_inherit,directory_inherit" a
sudo chmod +a "`whoami` allow delete,write,append,file_inherit,directory_inherit" a
```

Si vous rencontrez une erreur avec ces commandes, exécutez les commandes suivantes, qui n'utilisent pas le chmod +a :

Code : Console

```
sudo setfacl -R -m u:www-data:rwX -m u:`whoami`:rwX app/cache app/logs
sudo setfacl -dR -m u:www-data:rwX -m u:`whoami`:rwX app/cache app/logs
```

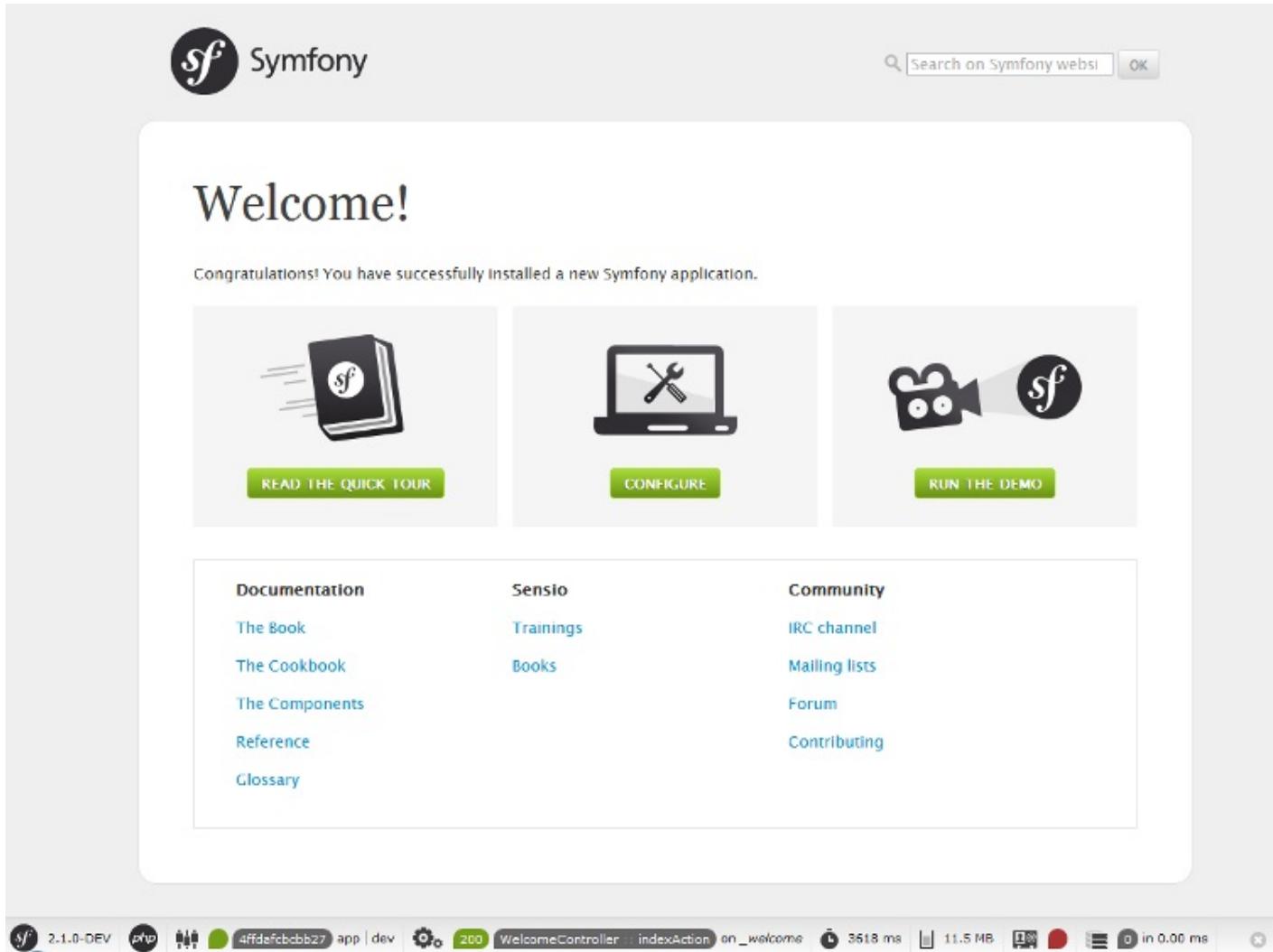
Enfin, si vous ne pouvez pas utiliser les ACL (utilisés dans les commandes précédentes), définissez simplement les droits comme suit :

Code : Console

```
chmod 777 app/cache  
chmod 777 app/logs
```

Voilà, fin de l'aparté.

Vous pouvez dès à présent exécuter Symfony2, félicitations ! Rendez-vous sur la page http://localhost/Symfony/web/app_dev.php, vous devriez avoir quelque chose ressemblant à la figure suivante.



Vérifier l'installation de PHP en console

Nous aurons parfois besoin d'exécuter des commandes PHP via la console, pour générer du code ou gérer la base de données. Ce sont des commandes qui vont nous faire gagner du temps (toujours le même objectif !), vérifions donc que PHP est bien disponible en console. Rassurez-vous, je vous indiquerai toujours pas à pas comment les utiliser. 😊

Si vous êtes sous Linux ou Mac, vous ne devriez pas avoir de soucis, PHP est bien disponible en console. Si vous êtes sous Windows, rien n'est sûr. Dans tous les cas, vérifiez-le en ouvrant l'invite de commandes pour Windows, ou le terminal pour Linux, et entrez la commande suivante : `php -v`. Si cette commande vous retourne bien la version de PHP et d'autres informations, c'est tout bon pour vous.

[La commande vous affiche une erreur ?](#)

Si vous êtes sous Windows, PHP est bien installé mais Windows ne sait pas où le trouver, il faut juste le lui dire. Voici la démarche à suivre pour régler ce problème :

1. Allez dans les paramètres système avancés
(Démarer > Panneau de configuration > Système et sécurité > Système > Paramètres système avancés);
2. Cliquez sur le bouton Variables d'environnement...;
3. Regardez dans le panneau Variables système;
4. Trouvez l'entrée Path (vous devriez avoir à faire descendre l'ascenseur pour le trouver);
5. Double-cliquez sur l'entrée Path;
6. Entrez votre répertoire PHP à la fin, sans oublier le point-virgule (;) auparavant. C'est le répertoire dans lequel se trouve le fichier php.exe. Par exemple ;C:\wamp\bin\php\php5.3;
7. Confirmez en cliquant sur OK. Vous devez ensuite redémarrer l'invite de commandes pour prendre en compte les changements.

Si vous êtes sous Linux, vérifiez votre installation de PHP. Vous devez notamment avoir le paquet php5-cli, qui est la version console de PHP.

Dans les deux cas, vérifiez après vos manipulations que le problème est bien résolu. Pour cela, exécutez à nouveau la commande `php -v`. Elle devrait alors vous afficher la version de PHP.

Et voilà, votre poste de travail est maintenant opérationnel pour développer avec Symfony2 !

En résumé

- Le mot « framework » signifie « cadre de travail » en français ;
- L'objectif principal d'un framework est d'améliorer la productivité des développeurs qui l'utilisent ;
- Contrairement aux CMS, un framework est destiné à des développeurs, et non à des novices en informatique ;
- L'apprentissage d'un framework est un investissement : il y a un certain effort à fournir au début, mais les résultats se récoltent ensuite sur le long terme !
- Symfony2 est un framework PHP très populaire, français, et très utilisé dans le milieu des entreprises.

Vous avez dit Symfony2 ?

Dans ce chapitre, nous allons voir comment est organisé Symfony2 à l'intérieur. Nous n'entrerons pas dans les détails, c'est trop tôt, le but est juste d'avoir une vision globale du processus d'exécution d'une page sous Symfony2. Ainsi, vous pourrez comprendre ce que vous faites. C'est mieux, non ? 

L'architecture des fichiers

On vient d'extraire beaucoup de fichiers, mais sans savoir encore à quoi ils servent. C'est le moment d'éclaircir tout cela !

Liste des répertoires

Ouvrez donc le répertoire dans lequel vous avez extrait les fichiers. Vous pouvez voir qu'il n'y a pas beaucoup de fichiers ici, seulement des répertoires. En effet, tout est bien rangé dans chaque répertoire, il nous faut donc comprendre à quoi ils servent. En voici la liste :

- app
- src
- vendor
- web

Le répertoire /app

Ce répertoire contient tout ce qui concerne votre site internet... sauf son code source. Assez étrange, me direz-vous. En fait, c'est simplement pour séparer le code source, qui fait la logique de votre site, du reste. Le reste, c'est ce répertoire /app. Et ce reste c'est : la configuration, le cache, les fichiers logs, etc. Ce sont des fichiers qui concernent l'entièreté de votre site, contrairement aux fichiers de code source qui seront découverts par fonctionnalité de votre site. Dans Symfony2, un projet de site internet est une **application**, simple question de vocabulaire. Le répertoire /app est donc le raccourci pour « application ».

Le répertoire /src

Voici enfin le répertoire dans lequel on mettra le code source ! C'est ici que l'on passera le plus clair de notre temps. Dans ce répertoire, nous organiserons notre code en *bundles*, des briques de notre application, dont nous verrons la définition plus loin.

Vous pouvez voir que ce répertoire n'est pas vide : il contient en effet quelques fichiers exemples, fournis par Symfony2. Nous les supprimerons plus tard dans ce cours.

Le répertoire /vendor

Ce répertoire contient toutes les bibliothèques externes à notre application. Dans ces bibliothèques externes, j'inclus Symfony2 ! Vous pouvez parcourir ce répertoire, vous y trouverez des bibliothèques comme Doctrine, Twig, SwiftMailer, etc.



Et une bibliothèque, c'est quoi exactement ?

Une bibliothèque est une sorte de boîte noire qui remplit une fonction bien précise, et dont on peut se servir dans notre code. Par exemple, la bibliothèque SwiftMailer permet d'envoyer des e-mails. On ne sait pas comment elle fonctionne (principe de la boîte noire), mais on sait comment s'en servir : on pourra donc envoyer des e-mails très facilement, juste en apprenant rapidement à utiliser la bibliothèque.

Le répertoire /web

Ce répertoire contient tous les fichiers destinés à vos visiteurs : images, fichiers CSS et JavaScript, etc. Il contient également le contrôleur frontal (app.php), dont nous parlerons juste après.

En fait, c'est le seul répertoire qui devrait être accessible à vos visiteurs. Les autres répertoires ne sont pas censés être accessibles (ce sont vos classes, elles vous regardent vous, pas vos visiteurs), c'est pourquoi vous y trouverez des fichiers .htaccess interdisant l'accès depuis l'extérieur. On utilisera donc toujours des URL du type [http://localhost/Symfony/web/...](http://localhost/Symfony/web/) au lieu de simplement [http://localhost/Symfony/....](http://localhost/Symfony/).



Si vous le souhaitez, vous pouvez configurer votre Apache pour que l'URL `http://localhost/Symfony` pointe directement sur le répertoire `/web`. Pour cela, vous pouvez lire [ce tutoriel qui explique comment configurer Apache](#). Cependant, ce n'est pas très important pour le développement, on en reparlera plus loin.

À retenir

Retenez donc que nous passerons la plupart de notre temps dans le répertoire `/src`, à travailler sur nos bundles. On touchera également pas mal au répertoire `/app` pour configurer notre application. Et lorsque nous installerons des bundles téléchargés, nous les ferons dans le répertoire `/vendor`.

Le contrôleur frontal

Définition

Le contrôleur frontal (*front controller*, en anglais) est le point d'entrée de votre application. C'est *le* fichier par lequel passent toutes vos pages. Vous devez sûrement connaître le principe d'`index.php` et des pseudo-frames (avec des URL du type `index.php?page=blog`) ; eh bien, cet `index.php` est un contrôleur frontal. Dans Symfony2, le contrôleur frontal se situe dans le répertoire `/web`, il s'agit de `app.php` ou `app_dev.php`.



Pourquoi y a-t-il deux contrôleurs frontaux ? Normalement, c'est un fichier unique qui gère toutes les pages, non ?

Vous avez parfaitement raison... pour un code classique ! Mais nous travaillons maintenant avec Symfony2, et son objectif est de nous faciliter le développement. C'est pourquoi Symfony2 propose un contrôleur frontal pour nos visiteurs, `app.php`, et un contrôleur frontal lorsque nous développons, `app_dev.php`. Ces deux contrôleurs frontaux, fournis par Symfony2 et prêts à l'emploi, définissent en fait deux environnements de travail.

Deux environnements de travail

L'objectif est de répondre au mieux suivant la personne qui visite le site :

- Un développeur a besoin d'informations sur la page afin de l'aider à développer. En cas d'erreur, il veut tous les détails pour pouvoir déboguer facilement. Il n'a pas besoin de rapidité.
- Un visiteur normal n'a pas besoin d'informations particulières sur la page. En cas d'erreur, l'origine de celle-ci ne l'intéresse pas du tout, il veut juste retourner d'où il vient. Par contre, il veut que le site soit le plus rapide possible à charger.

Vous voyez la différence ? À chacun ses besoins, et Symfony2 compte bien tous les remplir. C'est pourquoi il offre plusieurs environnements de travail :

- L'environnement de développement, appelé « dev », accessible en utilisant le contrôleur frontal `app_dev.php`. C'est l'environnement que l'on utilisera *toujours* pour développer.
- L'environnement de production, appelé « prod », accessible en utilisant le contrôleur frontal `app.php`.

Essayez-les ! Allez sur http://localhost/Symfony/web/app_dev.php et vous verrez une barre d'outils en bas de votre écran, contenant nombre d'informations utiles au développement. Allez sur <http://localhost/Symfony/web/app.php> et vous obtiendrez... une erreur 404. En effet, aucune page n'est définie par défaut pour le mode « prod ». Nous les définirons plus tard, mais notez que c'est une « belle » erreur 404, aucun terme barbare n'est employé pour la justifier.

Pour voir le comportement du mode « dev » en cas d'erreur, essayez aussi d'aller sur une page qui n'existe pas. Vous avez vu ce qui donne une page introuvable en mode « prod », mais allez maintenant sur [/app_dev.php/pagequandexistePas](#). La différence est claire : le mode « prod » nous dit juste « page introuvable » alors que le mode « dev » nous donne plein d'informations sur l'origine de l'erreur, indispensables pour la corriger.

C'est pourquoi, dans la suite du tutoriel, nous utiliserons toujours le mode « dev », en passant donc par `app_dev.php`. Bien sûr, lorsque votre site sera opérationnel et que des internautes pourront le visiter, il faudra leur faire utiliser le mode « prod ». Mais nous n'en sommes pas encore là.



Et comment savoir quelles erreurs surviennent en mode production si elles ne s'affichent pas ?

C'est une bonne question, en effet si par malheur une erreur intervient pour l'un de vos visiteurs, il ne verra aucun message et vous non plus, une vraie galère pour déboguer ! En réalité, si les erreurs ne sont pas affichées, elles sont bien stockées dans un fichier. Allez jeter un œil au fichier `app/logs/prod.log` qui contient plein d'informations sur les requêtes effectuées en mode production, dont les erreurs.

Concrètement, qu'est-ce que contrôle le contrôleur frontal ?

Très bonne question. Pour cela, rien de tel que... d'ouvrir le fichier `app.php`. Ouvrez-le et vous constaterez qu'il ne fait pas grand-chose. En effet, le but du contrôleur frontal n'est pas de *faire* quelque chose, mais d'être un *point d'entrée* de notre application. Il se limite donc à appeler le noyau (Kernel) de Symfony2 en disant « On vient de recevoir une requête, transforme-la en réponse s'il-te-plaît. »

Ici, voyez le contrôleur frontal comme un fichier à nous (il est dans notre répertoire `/web`), et le Kernel comme un composant Symfony2, une boîte noire (il est dans le répertoire `/vendor`). Vous voyez comment on a utilisé notre premier composant Symfony2 : on a délégué la gestion de la requête au Kernel. Bien sûr, ce Kernel aura besoin de nous pour savoir quoi exécuter comme code, mais il gère déjà plusieurs choses que nous avons vues : la gestion des erreurs, l'ajout de la `toolbar` en bas de l'écran, etc. On n'a encore rien fait, et pourtant on a déjà gagné du temps !

L'architecture conceptuelle

On vient de voir comment sont organisés les fichiers de Symfony2. Maintenant, il s'agit de comprendre comment s'organise l'exécution du code au sein de Symfony2.

Architecture MVC

Vous avez certainement déjà entendu parler de ce concept. Sachez que Symfony2 respecte bien entendu cette architecture MVC. Je ne vais pas entrer dans ses détails, car il y a déjà un [super cours sur le Site du Zéro](#), mais en voici les grandes lignes.

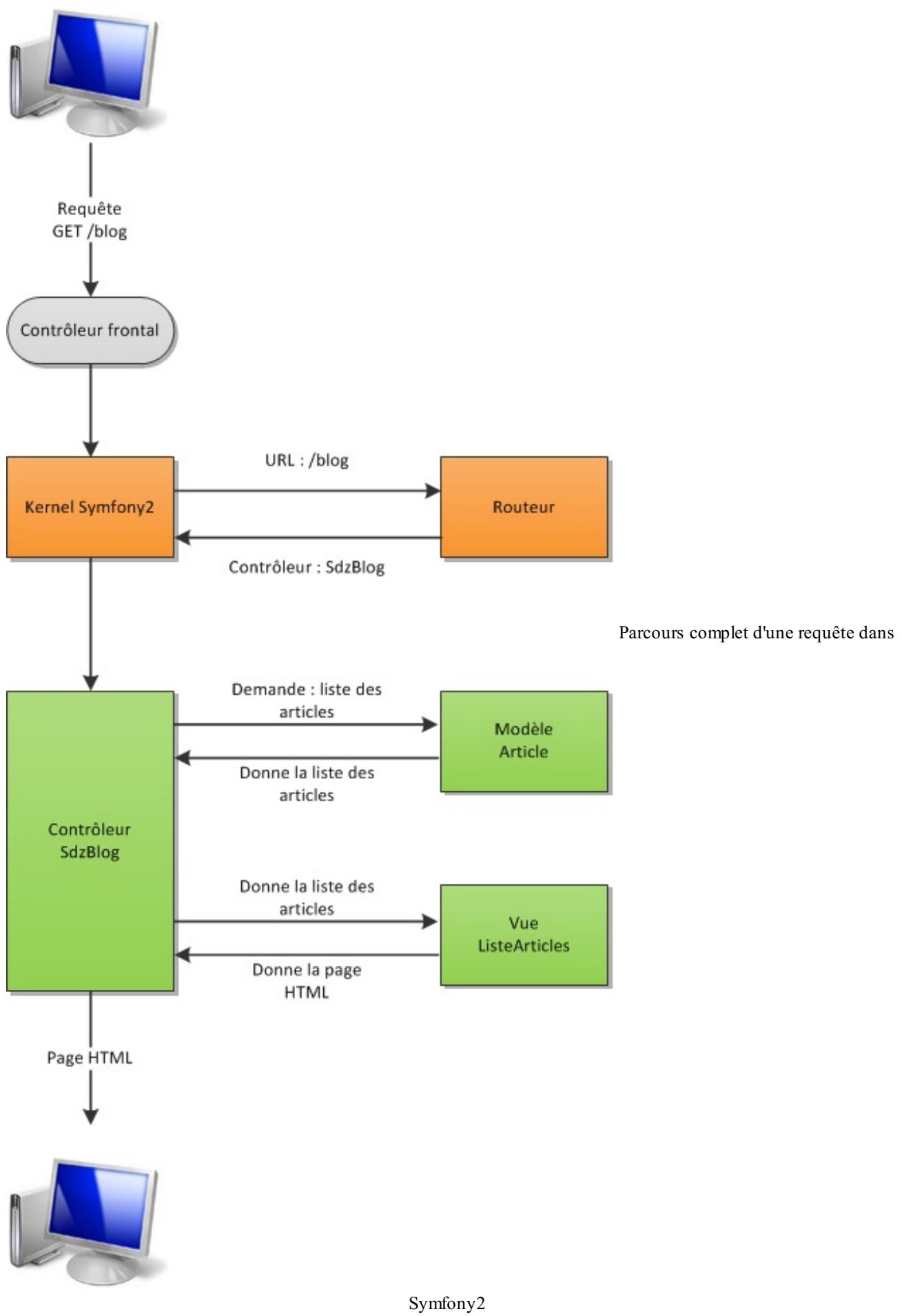
MVC signifie « Modèle / Vue / Contrôleur ». C'est un découpage très répandu pour développer les sites internet, car il sépare les couches selon leur logique propre :

- **Le Contrôleur** (ou *Controller*) : son rôle est de générer la réponse à la requête HTTP demandée par notre visiteur. Il est la couche qui se charge d'analyser et de traiter la requête de l'utilisateur. Le contrôleur contient la logique de notre site internet et va se contenter « d'utiliser » les autres composants : les modèles et les vues. Concrètement, un contrôleur va récupérer, par exemple, les informations sur l'utilisateur courant, vérifier qu'il a le droit de modifier tel article, récupérer cet article et demander la page du formulaire d'édition de l'article. C'est tout bête, avec quelques `if()`, on s'en sort très bien.
- **Le Modèle** (ou *Model*) : son rôle est de gérer vos données et votre contenu. Reprenons l'exemple de l'article. Lorsque je dis « le contrôleur récupère l'article », il va en fait faire appel au modèle `Article` et lui dire : « donne-moi l'article portant l'id 5 ». C'est le modèle qui sait comment récupérer cet article, généralement via une requête au serveur SQL, mais ce pourrait être depuis un fichier texte ou ce que vous voulez. Au final, il permet au contrôleur de manipuler les articles, mais sans savoir comment les articles sont stockés, gérés, etc. C'est une couche d'abstraction.
- **La Vue** (ou *View*) : son rôle est d'afficher les pages. Reprenons encore l'exemple de l'article. Ce n'est pas le contrôleur qui affiche le formulaire, il ne fait qu'appeler la bonne vue. Si nous avons une vue `Formulaire`, les balises HTML du formulaire d'édition de l'article y seront et au final le contrôleur ne fera qu'afficher cette vue sans savoir vraiment ce qu'il y a dedans. En pratique, c'est le designer d'un projet qui travaille sur les vues. Séparer vues et contrôleurs permet aux designers et développeurs PHP de travailler ensemble sans se marcher dessus.

Au final, si vous avez bien compris, le contrôleur ne contient que du code très simple, car il se contente d'utiliser des modèles et des vues en leur attribuant des tâches précises. Il agit un peu comme un chef d'orchestre, qui n'agit qu'une baguette alors que ses musiciens jouent des instruments complexes.

Parcours d'une requête dans Symfony2

Afin de bien visualiser tous les acteurs que nous avons vus jusqu'à présent, je vous propose un schéma du parcours complet d'une requête dans Symfony2 :



Symfony2

En le parcourant avec des mots, voici ce que cela donne :

1. Le visiteur demande la page /blog ;
2. Le contrôleur frontal reçoit la requête, charge le Kernel et la lui transmet ;
3. Le Kernel demande au Routeur quel contrôleur exécuter pour l'URL /blog. Ce Routeur est un composant Symfony2 qui fait la correspondance entre URL et contrôleurs, nous l'étudierons bien sûr dans un prochain chapitre. Le Routeur fait donc son travail, et dit au Kernel qu'il faut exécuter le contrôleur SdzBlog ;
4. Le Kernel exécute donc ce contrôleur. Le contrôleur demande au modèle Article la liste des articles, puis la donne à la vue ListeArticles pour qu'elle construise la page HTML et la lui retourne. Une fois cela fini, le contrôleur envoie au visiteur la page HTML complète.

J'ai mis des couleurs pour distinguer les points où l'on intervient. En vert donc, les contrôleur, modèle et vue, c'est ce qu'on devra développer nous-mêmes. En orange donc, le Kernel et le Routeur, c'est ce qu'on devra configurer. On ne touchera pas au contrôleur frontal, en gris.

Maintenant, il ne nous reste plus qu'à voir comment organiser concrètement notre code et sa configuration.

Symfony2 et ses bundles

La découpe en bundles

Le concept

Vous avez déjà croisé ce terme de bundle quelques fois depuis le début du cours, mais qu'est-ce qui se cache derrière ce terme ?

Pour faire simple, un bundle est une brique de votre application. Symfony2 utilise ce concept novateur qui consiste à regrouper dans un même endroit, le bundle, tout ce qui concerne une même fonctionnalité. Par exemple, on peut imaginer un bundle « Blog » dans notre site, qui regrouperait les contrôleurs, les modèles, les vues, les fichiers CSS et JavaScript, etc. Tout ce qui concerne directement la fonctionnalité blog de notre site.

Cette organisation permet de découper naturellement nos fonctionnalités, et ainsi de ranger chaque fichier à sa place. Un fichier JavaScript n'est utilisé que sur le bundle Blog ? Mettez-le dans le bundle Blog ! Bien évidemment, au sein d'un bundle, il faut retrouver également une architecture bien définie, nous l'étudierons juste après.

Des exemples

Pour mieux visualiser, je vous propose quelques bons exemples de bundles possibles :

- Un bundle Utilisateur, qui va gérer les utilisateurs ainsi que les groupes, intégrer des pages d'administration de ces utilisateurs, et des pages classiques comme le formulaire d'inscription, de récupération de mot de passe, etc.
- Un bundle Blog, qui va fournir une interface pour gérer un blog sur le site. Ce bundle peut utiliser le bundle Utilisateur pour faire un lien vers les profils des auteurs des articles et des commentaires.
- Un bundle Boutique, qui va fournir des outils pour gérer des produits et des commandes.
- Un bundle Admin, qui va fournir uniquement une interface vers les outils d'administration des différents bundles utilisés (Utilisateur, Blog, etc.). Attention, il ne doit pas y avoir beaucoup de code dans ce bundle, ce n'est qu'un raccourci vers les fonctionnalités d'administration des autres bundles. La partie admin pour ajouter un article au blog *doit* se trouver dans le bundle Blog.

Et ces bundles, parce qu'ils respectent des règles communes, vont fonctionner ensemble. Par exemple, un bundle Forum et un bundle Utilisateur devront s'entendre : dans un forum, ce sont des utilisateurs qui interagissent. 😊

L'intérêt

Une question à toujours se poser : quel est l'intérêt de ce que l'on est en train de faire ?

En plus d'organiser votre code par fonctionnalités, la découpe en bundles permet l'échange de bundles entre applications ! Cela signifie que vous pouvez développer une fonctionnalité, puis la partager avec d'autres développeurs ou encore la réutiliser dans un de vos autres projets. Et bien entendu, cela marche dans l'autre sens : vous pouvez installer dans votre projet des bundles qui ont été développés par d'autres !

Le principe même des bundles offre donc des possibilités infinies ! Imaginez le nombre de fonctionnalités classiques sur un site internet, que vous n'aurez plus à développer vous-mêmes. Vous avez besoin d'un livre d'or ? il existe sûrement un bundle. Vous

avez besoin d'un blog ? il existe sûrement un bundle, etc.

Les bundles de la communauté

Presque tous les bundles de la communauté Symfony2 sont regroupés sur un même site : <http://knxbundles.com/>. Il en existe beaucoup, et pour n'en citer que quelques-uns :

- **FOSUserBundle** : c'est un bundle destiné à gérer les utilisateurs de votre site. Concrètement, il fournit le modèle utilisateur ainsi que le contrôleur pour accomplir les actions de base (connexion, inscription, déconnexion, édition d'un utilisateur, etc.) et fournit aussi les vues qui vont avec. Bref, il suffit d'installer le bundle et de le personnaliser un peu pour obtenir un espace membre !
- **FOSCommentBundle** : c'est un bundle destiné à gérer des commentaires. Concrètement, il fournit le modèle commentaire (ainsi que son contrôleur) pour ajouter, modifier et supprimer les commentaires. Les vues sont fournies avec, évidemment. Bref, en installant ce bundle, vous pourrez ajouter un fil de commentaires à n'importe quelle page de votre site !
- **GravatarBundle** : c'est un bundle destiné à gérer les avatars depuis le service web [Gravatar](#). Concrètement, il fournit une extension au moteur de templates pour pouvoir afficher facilement un avatar issu de Gravatar via une simple fonction qui s'avère être très pratique.
- Etc.

Je vous conseille vivement de passer sur <http://knxbundles.com/> avant de commencer à développer un bundle. S'il en existe déjà un et qu'il vous convient, il serait trop bête de réinventer la roue. 😊 Bien sûr, il faut d'abord apprendre à installer un bundle externe, patience !

La structure d'un bundle

Un bundle contient tout : contrôleurs, vues, modèles, classes personnelles, etc. Bref, tout ce qu'il faut pour remplir la fonction du bundle. Évidemment, tout cela est organisé en dossiers afin que tout le monde s'y retrouve. Voici la structure d'un bundle à partir de son répertoire de base, vous pouvez en voir l'illustration grâce au bundle exemple fourni par défaut dans `src/Acme/DemoBundle/` :

Code : Console

```
/Controller           | Contient vos contrôleurs
/DependencyInjection | Contient des informations sur votre bundle (chargement autom
/Entity               | Contient vos modèles
/Form                | Contient vos éventuels formulaires
/Resources            |
-- /config            | Contient les fichiers de configuration de votre bundle (no
-- /public             | Contient les fichiers publics de votre bundle : fichiers C
-- /views              | Contient les vues de notre bundle, les templates Twig
/Tests                | Contient vos éventuels tests unitaires et fonctionnels. Nous
```

La structure est assez simple au final, retenez-la bien. Sachez qu'elle n'est pas du tout fixe, vous pouvez créer tous les dossiers que vous voulez pour mieux organiser votre code. Mais cette structure conventionnelle permet à d'autres développeurs de comprendre rapidement votre bundle. Bien entendu, je vous guiderai pour chaque création de fichier. 😊

En résumé

- Symfony2 est organisé en quatre répertoires : `app`, `src`, `vendor` et `web`.
- Le répertoire dans lequel on passera le plus de temps est `src`, il contient le code source de notre site.
- Il existe deux environnements de travail :
 - L'environnement « `prod` » est destiné à vos visiteurs : il est rapide à exécuter, et ne divulgue pas les messages d'erreur.
 - L'environnement « `dev` » est destiné au développeur, c'est-à-dire vous : il est plus lent, mais offre plein d'informations utiles au développement.
- Symfony2 utilise l'architecture MVC pour bien organiser les différentes parties du code source.
- Un bundle est une brique de votre application : il contient tout ce qui concerne une fonctionnalité donnée. Cela permet

de bien organiser les différentes parties de votre site.

- Il existe des milliers de bundles développés par la communauté, pensez à vérifier qu'il n'existe pas déjà un bundle qui fait ce que vous souhaitez faire !

Utilisons la console pour créer un bundle

Dans ce chapitre, nous allons créer notre premier bundle, juste histoire d'avoir la structure de base de notre code futur. Mais nous ne le ferons pas n'importe comment : nous allons générer le bundle en utilisant une commande Symfony2 en console ! L'objectif est de découvrir la console utilement.

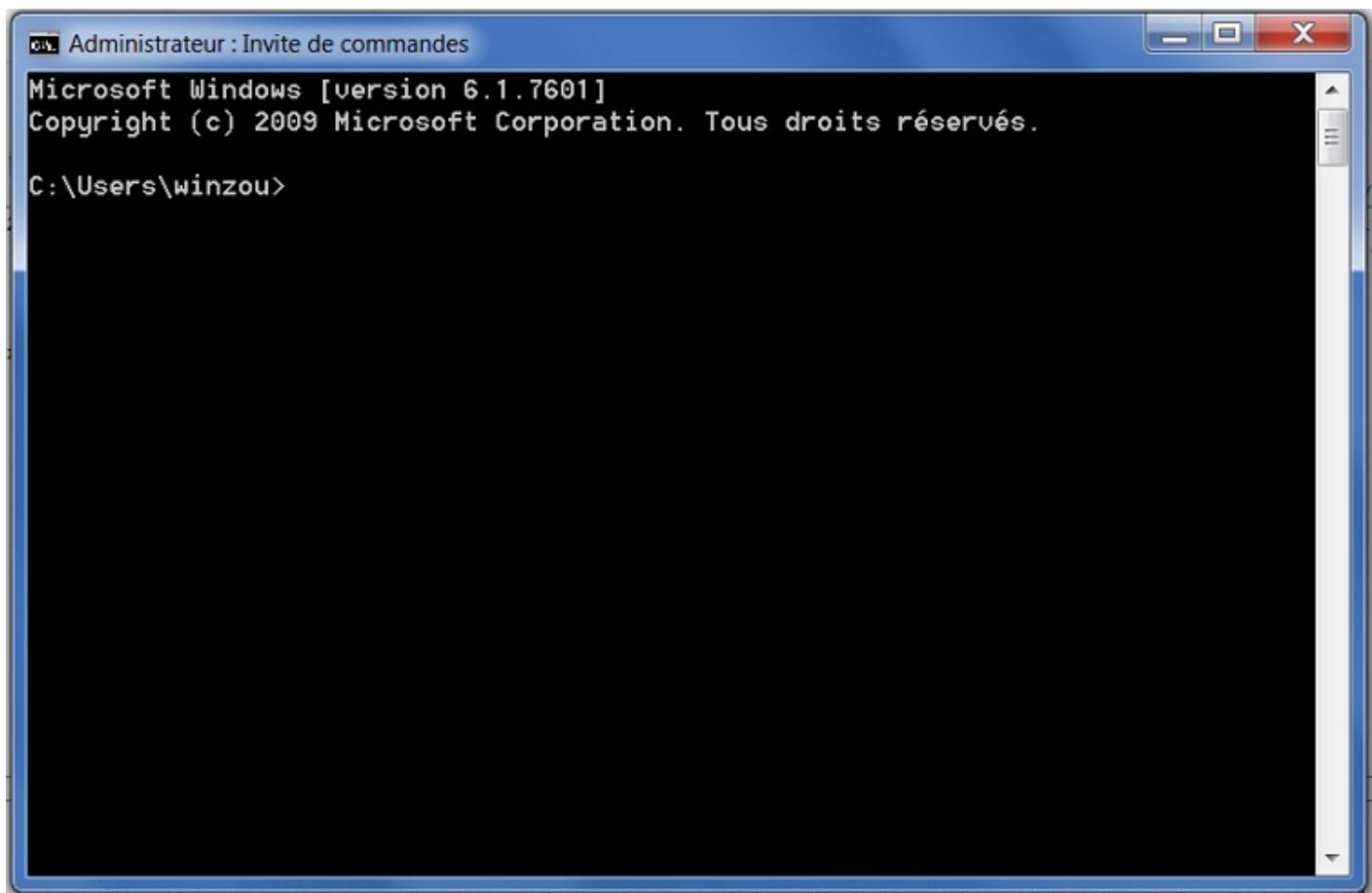
Utilisation de la console

Tout d'abord, vous devez savoir une chose : Symfony2 intègre des commandes disponibles non pas via le navigateur, mais via l'invite de commandes (sous Windows) ou le terminal (sous Linux). Il existe pas mal de commandes qui vont nous servir assez souvent lors du développement, apprenons donc dès maintenant à utiliser cette console !

Les outils disponibles en ligne de commande ont pour objectif de nous faciliter la vie. Ce n'est pas un obscur programme pour les geeks amoureux de la console ! Vous pourrez à partir de là générer une base de code source pour certains fichiers récurrents, vider le cache, ajouter des utilisateurs par la suite, etc. N'ayez pas peur de cette console.

Sous Windows

Lancez l'invite de commandes : Menu Démarrer > Programmes > Accessoires > Invite de commandes. Une fenêtre semblable à la figure suivante devrait apparaître.



La console Windows

Puis placez-vous dans le répertoire où vous avez mis Symfony2, en utilisant la commande Windows `cd` (je vous laisse adapter la commande) :

Code : Console

```
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Users\winzou>cd ../../wamp/www/Symfony

C:\wamp\www\Symfony>_
```



Vous avez également la possibilité de vous rendre dans le répertoire où vous avez mis Symfony2 via l'explorateur de fichiers et de faire un clic droit sur le dossier Symfony2 en appuyant en même temps sur la touche Shift de votre clavier. Dans le menu contextuel, choisissez Ouvrir une fenêtre de commandes ici.

On va exécuter des fichiers PHP depuis cette invite de commandes. En l'occurrence, c'est le fichier `app/console` (ouvrez-le, c'est bien du PHP) que nous allons exécuter. Pour cela, il faut lancer la commande PHP avec le nom du fichier en argument : `php app/console`. C'est parti :

Code : Console

```
C:\wamp\www\Symfony>php app/console  
Symfony version 2.2.0-RC2 - app/dev/debug  
  
Usage:  
  [options] command [arguments]  
  
Options:  
  --help          -h Display this help message.  
  --quiet         -q Do not output any message.  
  --verbose       -v Increase verbosity of messages.  
  --version       -V Display this application version.  
  --ansi          Force ANSI output.  
  --no-ansi        Disable ANSI output.  
  --no-interaction -n Do not ask any interactive question.  
  --shell          -s Launch the shell.  
  --process-  
isolation      Launch commands from shell as a separate processes.  
  --env            -e The Environment name.  
  --no-debug       Switches off debug mode.
```

Et voilà, vous venez d'exécuter une commande Symfony ! Celle-ci ne fait pas grand-chose, c'était juste un entraînement.



La commande ne fonctionne pas ? On vous dit que PHP n'est pas un exécutable ? Vous avez dû oublier d'ajouter PHP dans votre variable PATH, on l'a fait à la fin du premier chapitre, jetez-y un œil.

Sous Linux et Mac

Ouvrez le terminal. Placez-vous dans le répertoire où vous avez mis Symfony2, probablement `/var/www` pour Linux ou `/user/sites` pour Mac. Le fichier que nous allons exécuter est `app/console`, il faut donc lancer la commande `php app/console`. Je ne vous fais pas de capture d'écran, j'imagine que vous savez le faire !

À quoi ça sert ?

Une très bonne question, qu'il faut toujours se poser. 😊 La réponse est très simple : à nous simplifier la vie !

Depuis cette console, on pourra par exemple créer une base de données, vider le cache, ajouter ou modifier des utilisateurs (sans passer par phpMyAdmin !), etc. Mais ce qui nous intéresse dans ce chapitre, c'est la **génération de code**.

En effet, pour créer un bundle, un modèle ou un formulaire, le code de départ est toujours le même. C'est ce code-là que le générateur va écrire pour nous. Du temps de gagné !

Comment ça marche ?



Comment Symfony2, un framework pourtant écrit en PHP, peut-il avoir des commandes en console ?

Vous devez savoir que PHP peut s'exécuter depuis le navigateur, mais également depuis la console. En fait, côté Symfony2, tout est toujours écrit en PHP, il n'y a rien d'autre. Pour en être sûrs, ouvrez le fichier app/console :

Code : PHP

```
<?php

require_once __DIR__.'/bootstrap.php.cache';
require_once __DIR__.'/AppKernel.php';

use Symfony\Bundle\FrameworkBundle\Console\Application;
use Symfony\Component\Console\Input\ArgvInput;

$input = new ArgvInput();
$env = $input->getParameterOption(array('--env', '-e'),
getenv('SYMFONY_ENV') ?: 'dev');
$debug = getenv('SYMFONY_DEBUG') !== '0' && !$input-
>hasParameterOption(array('--no-debug', '')) && $env !== 'prod';

$kernel = new AppKernel($env, $debug);
$application = new Application($kernel);
$application->run();
```

Vous ne remarquez rien ? Il ressemble beaucoup au contrôleur frontal app.php ! En fait, il fait presque la même chose, il inclut les mêmes fichiers, et charge également le Kernel. Mais il définit la requête comme venant de la console, ce qui exécute du code différent par la suite. On pourra nous aussi écrire du code qui sera exécuté non pas depuis le navigateur (comme les contrôleurs habituels), mais depuis la console. Rien ne change pour le code, si ce n'est que l'affichage ne peut pas être en HTML bien évidemment.

Créons notre bundle

Tout est bundle

Rappelez-vous : dans Symfony2, chaque partie de votre site est un bundle. Pour créer notre première page, il faut donc d'abord créer notre premier bundle. Rassurez-vous, créer un bundle est extrêmement simple avec le générateur. Démonstration !

Exécuter la bonne commande

Comme on vient de l'apprendre, exécutez la commande `php app/console generate:bundle`.

1. Choisir le namespace

Symfony2 vous demande le namespace de votre bundle :

Code : Console

```
C:\wamp\www\Symfony>php app/console generate:bundle
```

```
Welcome to the Symfony2 bundle generator
```

```
Your application code must be written in bundles. This command helps
you generate them easily.
```

Each bundle is hosted under a namespace (like Acme/Bundle/BlogBundle). The namespace should begin with a "vendor" name like your company name, your project name, or your client name, followed by one or more optional category sub-namespaces, and it should end with the bundle name itself (which must have Bundle as a suffix).

See http://symfony.com/doc/current/cookbook/bundles/best_practices.html#index-1 for more details on bundle naming conventions.

Use / instead of \ for the namespace delimiter to avoid any problem.

Bundle namespace: _

Vous pouvez nommer votre namespace comme bon vous semble, il faut juste qu'il se termine par le suffixe « Bundle ». Par convention, on le compose de trois parties. Nous allons nommer notre namespace « Sdz\BlogBundle ». Explications :

1. « Sdz » est le namespace racine : il vous représente. Vous pouvez mettre votre pseudo, le nom de votre site ou ce que vous voulez ;
2. « Blog » est le nom du bundle en lui-même : il définit ce que fait le bundle. Ici, nous créons un blog, nous l'avons donc simplement appelé « Blog » ;
3. « Bundle » est le suffixe obligatoire.

Entrez donc dans la console `Sdz/BlogBundle`, avec des slashes juste pour cette fois pour les besoins de la console, mais un namespace comprend bien des anti-slashes.

2. Choisir le nom

Symfony2 vous demande le nom de votre bundle :

Code : Console

Bundle namespace: Sdz/BlogBundle

In your code, a bundle is often referenced by its name. It can be the concatenation of all namespace parts but it's really up to you to come up with a unique name (a good practice is to start with the vendor name). Based on the namespace, we suggest SdzBlogBundle.

Bundle name [SdzBlogBundle]: _

Par convention, on nomme le bundle de la même manière que le namespace, sans les slashes. On a donc : SdzBlogBundle. C'est ce que Symfony2 vous propose par défaut (la valeur entre les crochets), appuyez donc simplement sur Entrée. Retenez ce nom : par la suite, quand on parlera du nom du bundle, cela voudra dire ce nom-là : SdzBlogBundle.

3. Choisir la destination

Symfony2 vous demande l'endroit où vous voulez que les fichiers du bundle soient générés :

Code : Console

The bundle can be generated anywhere. The suggested default directory uses the standard conventions.

Target directory [C:/wamp/www/src]: _

Par convention, comme on l'a vu, on place nos bundles dans le répertoire `/src`. C'est ce que Symfony2 vous propose, appuyez donc sur Entrée.

4. Choisir le format de configuration

Symfony2 vous demande sous quelle forme vous voulez configurer votre bundle. Il s'agit simplement du format de la configuration, que nous ferons plus tard. Il existe plusieurs moyens comme vous pouvez le voir : YAML, XML, PHP ou Annotations.

Code : Console

```
Target directory [C:/wamp/www/src] :
```

```
Determine the format to use for the generated configuration.
```

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

Chacun a ses avantages et inconvénients. Nous allons utiliser le YAML (`yml`) ici, car il est bien adapté pour un bundle. Mais sachez que nous utiliserons les annotations pour nos futures entités par exemple. Entrez donc `yml`.

5. Choisir quelle structure générer

Symfony2 vous demande si vous voulez générer juste le minimum ou une structure plus complète pour le bundle :

Code : Console

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

```
To help you get started faster, the command can generate some code snippets for you.
```

```
Do you want to generate the whole directory structure [no] ?
```

Faisons simple et demandons à Symfony2 de tout nous générer. Entrez donc `yes`.

6. Confirmez, et c'est joué !

Pour toutes les questions suivantes, confirmez en appuyant sur Entrée à chaque fois. Et voilà, votre bundle est généré :

Code : Console

```
Do you want to generate the whole directory structure [no] ? yes
```

```
Summary before generation
```

```
You are going to generate a "Sdz\BlogBundle\SdzBlogBundle" bundle  
in "C:/wamp/www/Symfony/src/" using the "yml" format.
```

```
Do you confirm generation [yes] ?
```

```
Bundle generation
```

```
Generating the bundle code: OK
Checking that the bundle is autoloaded: OK
Confirm automatic update of your Kernel [yes]?
Enabling the bundle inside the Kernel: OK
Confirm automatic update of the Routing [yes]?
Importing the bundle routing resource: OK
```

You can now start using the generated code!

C:\wamp\www\Symfony>_

 Tout d'abord, je vous réserve une petite surprise : allez voir sur http://localhost/Symfony/web/app_dev.php/hello/winzou ! Le bundle est déjà opérationnel ! Le code-exemple généré affiche le texte passé dans l'URL, vous pouvez donc également essayer ceci : http://localhost/Symfony/web/app_dev.php/hello/World.

 Mais pourquoi n'y a-t-il pas la *toolbar* en bas de la page ?

C'est normal, c'est juste un petit truc à savoir pour éviter de s'arracher les cheveux inutilement.  La *toolbar* est un petit bout de code HTML que rajoute Symfony2 à chaque page... contenant la balise `</body>`. Or sur cette page, vous pouvez afficher la source depuis votre navigateur, il n'y a aucune balise HTML en fait, donc Symfony2 n'ajoute pas la *toolbar*.

Pour l'activer, rien de plus simple, il nous faut rajouter une toute petite structure HTML. Pour cela, ouvrez le fichier `src/Sdz/BlogBundle/Resources/views/Default/index.html.twig`, c'est la vue utilisée pour cette page. L'extension `.twig` signifie qu'on utilise le moteur de templates Twig pour gérer nos vues, on en reparlera bien sûr. Le fichier est plutôt simple, et je vous propose de le changer ainsi :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Default/index.html.twig #-}

<html>
  <body>
    Hello {{ name }}!
  </body>
</html>
```

Actualisez la page, et voici une magnifique *toolbar* semblable à la figure suivante qui apparaît en bas de la page ! Seule la balise `</body>` suffisait, mais quitte à changer autant avoir une structure HTML valide. 



Que s'est-il passé ?

Dans les coulisses, Symfony2 a fait pas mal de choses, revoyons tout cela à notre rythme.

Symfony2 a généré la structure du bundle

Allez dans le répertoire `src/Sdz/BlogBundle`, vous pouvez voir tout ce que Symfony2 a généré pour nous. Rappelez-vous la structure d'un bundle que nous avons vu au chapitre précédent : Symfony2 en a généré la plus grande partie !

À savoir : le seul fichier obligatoire pour un bundle est en fait la classe `SdzBlogBundle.php` à la racine du répertoire. Vous pouvez l'ouvrir et voir ce qu'il contient : pas très intéressant en soi ; heureusement que Symfony l'a généré tout seul. Sachez-le dès maintenant : nous ne modifierons presque jamais ce fichier, vous pouvez passer votre chemin.

Symfony2 a enregistré notre bundle auprès du Kernel

Le bundle est créé, mais il faut dire à Symfony2 de le charger. Pour cela il faut configurer le noyau (le Kernel) pour qu'il le charge. Rappelez-vous, la configuration de l'application se trouve dans le répertoire `/app`. En l'occurrence, la configuration du noyau se fait dans le fichier `app/AppKernel.php` :

Code : PHP

```
<?php

use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new Symfony\Bundle\SecurityBundle\SecurityBundle(),
            new Symfony\Bundle\TwigBundle\TwigBundle(),
            new Symfony\Bundle\MonologBundle\MonologBundle(),
            new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
            new Symfony\Bundle\AsseticBundle\AsseticBundle(),
            new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
            new
Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
            new JMS\AopBundle\JMSAopBundle(),
            new JMS\DiExtraBundle\JMSDiExtraBundle($this),
            new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
            new Sdz\BlogBundle\SdzBlogBundle(), // Le générateur a rajouté
cette ligne
        );

        if (in_array($this->getEnvironment(), array('dev', 'test'))) {
            $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
            $bundles[] = new
Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
            $bundles[] = new
Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
            $bundles[] = new
Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
        }

        return $bundles;
    }

    // ...
}
```

Cette classe permet donc uniquement de définir quels bundles charger pour l'application. Vous pouvez le voir, ils sont instanciés dans un simple tableau. Les lignes 11 à 22 définissent les bundles à charger pour l'environnement de production. Les lignes 26 à 29 définissent les bundles à charger *en plus* pour l'environnement de développement.

Comme vous pouvez le voir, le générateur du bundle a modifié lui-même ce fichier pour y ajouter la ligne 22. C'est ce que l'on appelle « enregistrer le bundle dans l'application ».

Vous pouvez voir également qu'il en existe plein d'autres, ce sont tous les bundles par défaut qui apportent des fonctionnalités de base au framework Symfony2. En fait, quand on parle de Symfony2, on parle à la fois de ses composants (Kernel, Routeur, etc.) et de ses bundles.

Symfony2 a enregistré nos routes auprès du Routeur



Les routes ? Le Routeur ?

Pas de panique, nous verrons tout cela dans les prochains chapitres. Sachez juste pour l'instant que le rôle du Routeur, que nous avons brièvement vu sur le schéma du chapitre précédent, est de déterminer quel contrôleur exécuter en fonction de l'URL appelée. Pour cela, il utilise les routes.

Chaque bundle dispose de ses propres routes. Pour notre bundle fraîchement créé, vous pouvez les voir dans le fichier `src/Sdz/BlogBundle/Resources/config/routing.yml`. En l'occurrence il n'y en a qu'une seule :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
SdzBlogBundle_homepage:
    path:      /hello/{name}
    defaults: { _controller: SdzBlogBundle:Default:index }
```

Or ces routes ne sont pas chargées automatiquement, il faut dire au Routeur « Bonjour, mon bundle SdzBlogBundle contient des routes qu'il faut que tu viennes chercher. » Cela se fait, vous l'aurez deviné, dans la configuration de l'application. Cette configuration se trouve toujours dans le répertoire `/app`, en l'occurrence pour les routes il s'agit du fichier `app/config/routing.yml` :

Code : YAML

```
# app/config/routing.yml
SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix:   /
```

Ce sont ces lignes qui importent le fichier de routes situé dans notre bundle. Ces lignes ont déjà été générées par le générateur de bundle, vraiment pratique, lui !

À retenir

Ce qu'il faut retenir de tout cela, c'est que pour qu'un bundle soit opérationnel il faut :

- Son code source, situé dans `src/Application/Bundle`, et dont le seul fichier obligatoire est la classe à la racine `SdzBlogBundle.php` ;
- Enregistrer le bundle dans le noyau pour qu'il soit chargé, en modifiant le fichier `app/AppKernel.php` ;
- Enregistrer les routes (si le bundle en contient) dans le Routeur pour qu'elles soient chargées, en modifiant le fichier `app/config/routing.yml`.

Ces trois points sont bien sûr effectués automatiquement lorsqu'on utilise le générateur. Mais vous pouvez tout à fait créer un bundle sans l'utiliser, et il faudra alors remplir cette petite *checklist*.

Par la suite, tout notre code source sera situé dans des bundles. Un moyen très propre de bien structurer son application.

En résumé

- Les commandes Symfony2 disponibles en ligne de commande ont pour objectif de nous faciliter la vie en automatisant certaines tâches ;
- Les commandes sont faites, comme tout Symfony2, en PHP uniquement. La console n'est qu'un moyen différent du navigateur pour exécuter du code PHP ;
- La commande pour générer un nouveau bundle est `php app/console generate:bundle`.

Partie 2 : Les bases de Symfony2

Cette partie a pour objectif de créer une première page, et d'en maîtriser les composantes. Ce sont ici les notions les plus importantes de Symfony2, prenez donc votre temps pour bien comprendre tout ce qui est abordé.

Mon premier « Hello World ! » avec Symfony2

L'objectif de ce chapitre est de créer de toutes pièces notre première page avec Symfony2 : une simple page blanche comprenant un « Hello World ! ». Nous allons donc créer tous les éléments indispensables pour concevoir une telle page.

Nous allons avoir une vue d'ensemble de tous les acteurs qui interviennent dans la création d'une page : routeur, contrôleur et template. Pour cela, tous les détails ne seront pas expliqués afin qu'on se concentre sur l'essentiel : la façon dont ils se coordonnent. Vous devrez attendre les prochains chapitres pour étudier un à un ces trois acteurs, patience donc !

Ne bloquez donc pas sur un point si vous ne comprenez pas tout, forcez-vous juste à comprendre l'ensemble. À la fin du chapitre, vous aurez une vision globale de la création d'une page et l'objectif sera atteint.

Bonne lecture !

Créons notre route

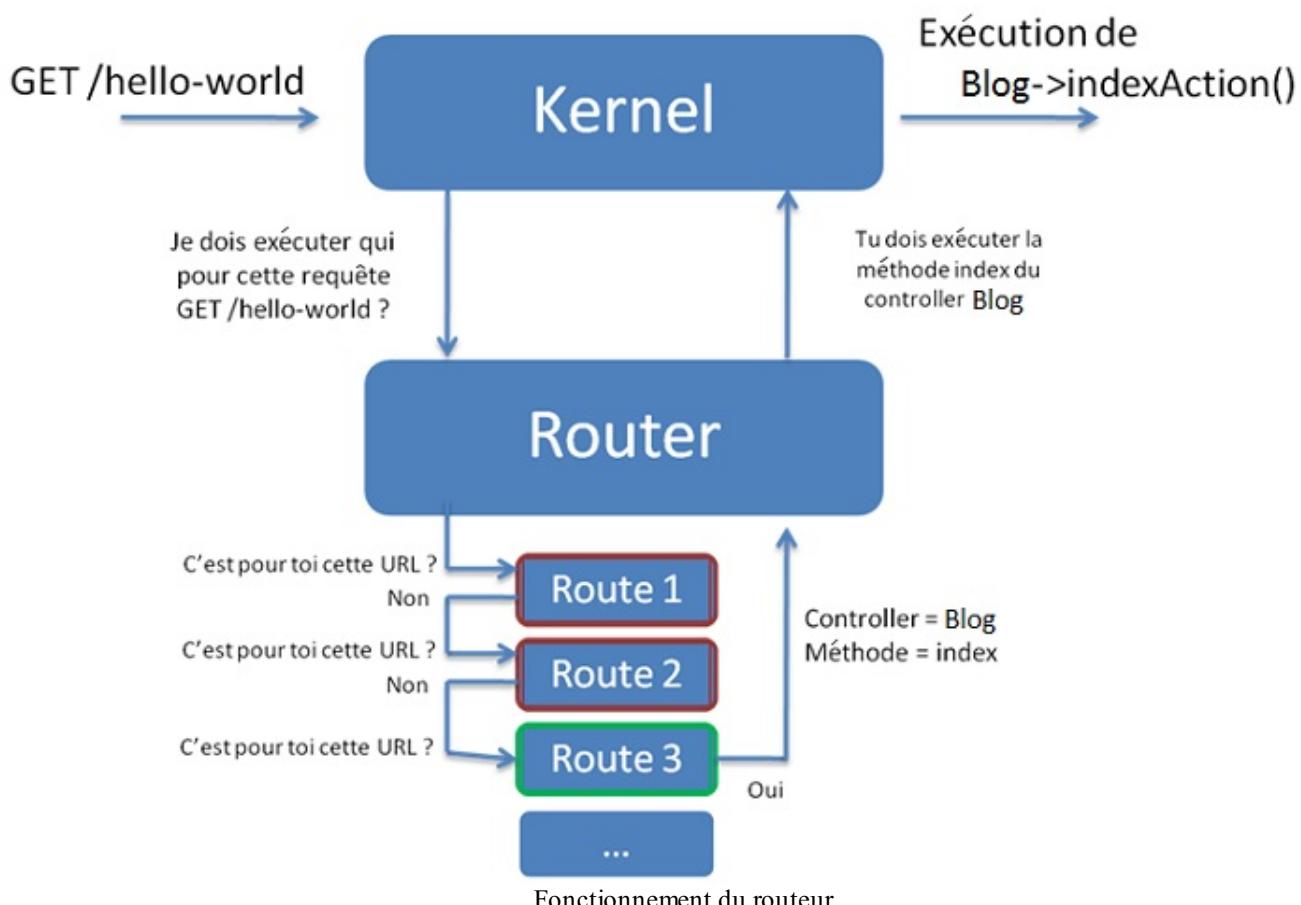
Nous travaillons dans notre bundle SdzBlogBundle, placez-vous donc dans son répertoire : `src/Sdz/BlogBundle`.

Pour créer une page, il faut d'abord définir l'URL à laquelle elle sera accessible. Pour cela, il faut créer la route de cette page.

Le routeur (ou router) ? Une route ?

Objectif

L'objectif du routeur est de dire à Symfony2 ce qu'il doit faire lorsque l'on appelle l'URL `/hello-world` (par exemple). Nous devons donc créer une route qui va dire : « Lorsque l'on est sur l'URL `/hello-world`, alors on appelle le contrôleur "Blog" qui va afficher un "Hello World !" ». Regardez la figure suivante.



Comme je l'ai dit, nous ne toucherons ni au noyau, ni au routeur : nous nous occuperons juste des routes.

1. Créons notre fichier de routes

Les routes se définissent dans un simple fichier texte, que Symfony2 a déjà généré pour notre SdzBlogBundle. Usuellement, on nomme ce fichier `Resources/config/routing.yml` dans le répertoire du bundle. Ouvrez le fichier, et ajoutez cette route à la suite de celle qui existe déjà :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
HelloTheWorld:
    path:      /hello-world
    defaults: { _controller: SdzBlogBundle:Blog:index }
```

Vous venez de créer votre première route !



Contrairement à ce que vous pourriez penser, l'indentation se fait avec 4 espaces par niveau, *pas avec des tabulations !* Je le précise parce qu'un jour vous ferez l'erreur (l'inévitable ne peut être évité), et vous me remercierez de vous avoir mis sur la voie. Et cela est valable pour tous vos fichiers `.yml`.



Attention également, il semble y avoir des erreurs lors des copier-coller depuis le tutoriel vers les fichiers `.yml`. Si vous rencontrez une obscure erreur, pensez à bien définir l'encodage du fichier en « UTF-8 sans BOM » et à supprimer les éventuels caractères non désirés. C'est un bug étrange qui provient du site, mais dont on ne connaît pas l'origine. L'esquive est de toujours recopier les exemples YAML que je vous donne, et de ne pas les copier-coller.



Si vous utilisez une version antérieure à Symfony 2.2, c'est « pattern » qu'il vous faut utiliser en lieu et place du « path » qu'on a utilisé ici. Cela ne change strictement rien au comportement, c'est juste le nom qui a changé.

Essayons de comprendre rapidement cette route :

- `HelloTheWorld` est le nom de la route. Il est assez arbitraire, et vous permet juste de vous y retrouver par la suite. La seule contrainte est qu'il soit unique.
- `path` correspond à l'URL à laquelle nous souhaitons que notre « Hello World ! » soit accessible. C'est ce qui permet à la route de dire : « Cette URL est pour moi, je prends. »
- `defaults` correspond aux paramètres de la route, dont :
 - `_controller`, qui correspond à l'**action** (ici, « index ») que l'on veut exécuter et au **contrôleur** (ici, « Blog ») que l'on va appeler (un contrôleur peut contenir plusieurs actions, c'est-à-dire plusieurs pages).

Ne vous inquiétez pas, un chapitre complet est consacré au routeur et vous permettra de jouer avec. Pour l'instant ce fichier nous permet juste d'avancer.

Mais avant d'aller plus loin, penchons-nous sur la valeur que l'on a donnée à `_controller` : « `SdzBlogBundle:Blog:index` ». Cette valeur se découpe en suivant les deux-points (`::`) :

- « `SdzBlogBundle` » est le nom de notre bundle, celui dans lequel Symfony2 ira chercher le contrôleur.
- « `Blog` » est le nom du contrôleur à ouvrir. En terme de fichier, cela correspond à `controller/BlogController.php` dans le répertoire du bundle. Dans notre cas, nous avons `src/Sdz/BlogBundle/controller/BlogController.php` comme chemin absolu.
- « `index` » est le nom de la méthode à exécuter au sein du contrôleur.

2. Informons Symfony2 que nous avons des routes pour lui

On l'a vu précédemment, grâce au bon travail du générateur, Symfony2 est déjà au courant du fichier de routes de notre bundle. Mais ce n'est pas par magie ! Il faut que vous sachiez comment tout cela s'imbrique. Ouvrez le fichier de configuration globale de notre application : `app/config/config.yml`. Dans ce fichier, il y a plein de valeurs, mais la section qui nous intéresse est la section `router`, à la ligne 9 que je vous remets ici :

Code : YAML

```
# app/config/config.yml

router:
    resource: "%kernel.root_dir%/config/routing.yml"
    strict_parameters: %kernel.debug%
```

Cette section indique au routeur qu'il doit chercher les routes dans le fichier `app/config/routing.yml` (`%kernel.root_dir%` est un paramètre qui vaut « `app` » dans notre cas). Le routeur va donc se contenter d'ouvrir ce fichier. Ouvrez-le également :

Code : YAML

```
# app/config/routing.yml

SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix:   /
```

Outre les commentaires, vous voyez que le générateur a inséré une route spéciale (qui n'a pas de path, mais une `resource`) qui va importer le fichier de routes de notre bundle.

 C'est parce que ce fichier `routing.yml` était vide (avant la génération du bundle) que l'on avait une erreur « page introuvable » en « prod » : comme il n'y a aucune route définie, Symfony2 nous informe à juste titre qu'aucune page n'existe. Et si le mode « dev » ne nous donnait pas d'erreur, c'est parce qu'il charge le fichier `routing_dev.yml` et non `routing.yml`. Et dans ce fichier, allez voir, il y a bien quelques routes définies. Et il y a aussi la ligne qui importe le fichier `routing.yml`, afin d'avoir les routes du mode « prod » dans le mode « dev » (l'inverse étant bien sûr faux).

Bref, vous n'avez rien à modifier ici, c'était juste pour que vous sachiez que l'import du fichier de routes d'un bundle n'est pas automatique, il se définit dans le fichier de routes global.

Revenons à nos moutons. En fait, on aurait pu ajouter notre route `HelloTheWorld` directement dans ce fichier `routing.yml`. Cela aurait fonctionné et cela aurait été plutôt rapide. Mais c'est oublier notre découpage en bundles ! En effet, cette route concerne le bundle du blog, elle doit donc se trouver dans notre bundle et pas ailleurs. N'oubliez jamais ce principe.

Cela permet à notre bundle d'être indépendant : si plus tard nous ajoutons, modifions ou supprimons des routes dans notre bundle, nous ne toucherons qu'au fichier `src/Sdz/BlogBundle/Resources/config/routing.yml` au lieu de `app/config/routing.yml`. 

Et voilà, il n'y a plus qu'à créer le fameux contrôleur Blog ainsi que sa méthode `index` !

Créons notre contrôleur

Le rôle du contrôleur

Rappelez-vous ce que nous avons dit sur le MVC :

- Le contrôleur est la « glu » de notre site ;
- Il « utilise » tous les autres composants (base de données, formulaires, templates, etc.) pour générer la réponse suite à notre requête ;
- C'est ici que résidera toute la logique de notre site : si l'utilisateur est connecté et qu'il a le droit de modifier cet article, alors j'affiche le formulaire d'édition des articles de mon blog.

Créons notre contrôleur

1. Le fichier de notre contrôleur Blog

Dans un bundle, les contrôleurs se trouvent dans le répertoire `Controller` du bundle. Rappelez-vous : dans la route, on a dit qu'il fallait faire appel au contrôleur nommé « Blog ». Le nom des fichiers des contrôleurs doit respecter une convention très simple : il doit commencer par le nom du contrôleur, ici « Blog », suivi du suffixe « Controller ». Au final, on doit donc créer le fichier `src/Sdz/BlogBundle/Controller/BlogController.php`.

Même si Symfony2 a déjà créé un contrôleur `DefaultController` pour nous, ce n'est qu'un exemple, on va utiliser le nôtre. Ouvrez donc notre `BlogController.php` et mettez-y le code suivant :

Code : PHP

```
<?php

// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function indexAction()
    {
        return new Response("Hello World !");
    }
}
```

Surprise : allez voir sur http://localhost/Symfony/web/app_dev.php/hello-world ! Même bundle, mais contrôleur différent, on en fait des choses !

Hello World ! Le Hello World s'affiche !

Maintenant, essayons de comprendre rapidement ce fichier :

- Ligne 5 : on se place dans le namespace des contrôleurs de notre bundle. Rien de bien compliqué, suivez la structure des répertoires dans lequel se trouve le contrôleur.
- Ligne 7 : notre contrôleur hérite de ce contrôleur de base, il faut donc le charger grâce au `use`.
- Ligne 8 : notre contrôleur va utiliser l'objet `Response`, il faut donc le charger grâce au `use`.
- Ligne 10 : le nom de notre contrôleur respecte le nom du fichier pour que l'`autoload` fonctionne.
- Ligne 12 : on définit la méthode `indexAction()`. N'oubliez pas de mettre le suffixe `Action` derrière le nom de la méthode.
- Ligne 14 : on crée une réponse toute simple. L'argument de l'objet `Response` est le contenu de la page que vous envoyez au visiteur, ici « Hello World ! ». Puis on retourne cet objet.

Bon, certes, le rendu n'est pas très joli, mais au moins nous avons atteint l'objectif d'afficher nous-mêmes un « Hello World ! ».



Pourquoi `indexAction()` ? Je n'ai pas suivi, là.

En effet, il faut savoir que le nom des méthodes des contrôleurs doit respecter une convention. Lorsque, dans la route, on parle de l'action « index », dans le contrôleur on doit définir la méthode `indexAction()`, c'est-à-dire le nom de l'action suivi du suffixe « Action », tout simplement. Il n'y a pas tellement à réfléchir, c'est une simple convention pour distinguer les méthodes

qui vont être appelées par le noyau (les `xxxAction()`) des autres méthodes que vous pourriez créer au sein de votre contrôleur.

Mais écrire le contenu de sa page de cette manière dans le contrôleur, ce n'est pas très pratique, et en plus de cela on ne respecte pas le modèle MVC. Utilisons donc les templates !

Créons notre template Twig

Les templates avec Twig

Savez-vous ce qu'est un moteur de templates ? C'est un script qui permet d'utiliser des templates, c'est-à-dire des fichiers qui ont pour but d'afficher le contenu de votre page HTML de façon dynamique, mais sans PHP. Comment ? Avec leur langage à eux. Chaque moteur a son propre langage.

Avec Symfony2, nous allons employer le moteur Twig. Voici un exemple de comparaison entre un template simple en PHP (premier code) et un template en « langage Twig » (deuxième code).

Code : PHP

```
<!DOCTYPE html>
<html>
    <head>
        <title>Bienvenue dans Symfony2 !</title>
    </head>
    <body>
        <h1><?php echo $titre_page; ?></h1>

        <ul id="navigation">
            <?php foreach ($navigation as $item) { ?>
                <li>
                    <a href="php echo $item-&gt;getHref(); ?&gt;"&gt;&lt;?php echo
$item-&gt;getTitre(); ?&gt;&lt;/a&gt;
                &lt;/li&gt;
            &lt;?php } ?&gt;
        &lt;/ul&gt;
    &lt;/body&gt;
&lt;/html&gt;</pre

```

Code : HTML & Django

```
<!DOCTYPE html>
<html>
    <head>
        <title>Bienvenue dans Symfony2 !</title>
    </head>
    <body>
        <h1>{{ titre_page }}</h1>

        <ul id="navigation">
            {%
                for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.titre }}</a></li>
            {%
                endfor %}
        </ul>
    </body>
</html>
```

Ils se ressemblent, soyons d'accord. Mais celui réalisé avec Twig est bien plus facile à lire ! Pour afficher une variable, vous faites juste `{{ ma_var }}` au lieu de `<?php echo $ma_var; ?>`.

Le but en fait est de faciliter le travail de votre designer. Un designer ne connaît pas forcément le PHP, ni forcément Twig d'ailleurs. Mais Twig est très rapide à prendre en main, plus rapide à écrire et à lire, et il dispose aussi de fonctionnalités très intéressantes. Par exemple, imaginons que votre designer veuille mettre les titres en lettres majuscules (COMME CECI). Il lui suffit de faire : `{{ titre|upper }}`, où `titre` est la variable qui contient le titre d'un article de blog par exemple. C'est plus

joli que <?php echo strtoupper(\$titre); ?>, non ?

Nous verrons dans le chapitre dédié à Twig les nombreuses fonctionnalités que le moteur vous propose et qui vont vous faciliter la vie. En attendant, nous devons avancer sur notre « Hello World ! ».

Utiliser Twig avec Symfony2

Comment utiliser un template Twig depuis notre contrôleur, au lieu d'afficher notre texte tout simple ?

1. Créons le fichier du template

Le répertoire des templates (ou vues) d'un bundle se trouve dans le dossier `Resources/views`. Ici encore, on ne va pas utiliser le template situé dans le répertoire `Default` généré par Symfony2. Créons notre propre répertoire `Blog` et créons notre template `index.html.twig` dans ce répertoire. Nous avons donc le fichier `src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig`.

Je vous propose de découper ce nom `Blog/index.html.twig` :

- `Blog` / est le nom du répertoire. Nous l'avons appelé comme notre contrôleur afin de nous y retrouver (ce n'est pas une obligation, mais c'est fortement recommandé).
- `index` est le nom de notre template qui est aussi le nom de la méthode de notre contrôleur (*idem*, pas obligatoire, mais recommandé).
- `html` correspond au format du contenu de notre template. Ici, nous allons y mettre du code HTML, mais vous serez amené à vouloir y mettre du XML ou autre : vous changerez donc cette extension. Cela permet de mieux s'y retrouver.
- `twig` est le format de notre template. Ici, nous utilisons Twig comme moteur de templates, mais il est toujours possible d'utiliser des templates PHP.

Revenez à notre template et mettez ce code à l'intérieur :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #-}

<!DOCTYPE html>
<html>
  <head>
    <title>Bienvenue sur ma première page avec le Site du Zéro
  </title>
  </head>
  <body>
    <h1>Hello World !</h1>

    <p>
      Le Hello World est un grand classique en programmation.
      Il signifie énormément, car cela veut dire que vous avez
      réussi à exécuter le programme pour accomplir une tâche simple
    :
      afficher ce hello world !
    </p>
  </body>
</html>
```

Dans ce template, nous n'avons utilisé ni variable, ni structure Twig. En fait, c'est un simple fichier contenant du code HTML pur !

2. Appelons ce template depuis le contrôleur

Il ne reste plus qu'à appeler ce template. C'est le rôle du contrôleur, c'est donc au sein de la méthode `indexAction()` que

nous allons appeler le template. Cela se fait très simplement avec la méthode `$this->render()`. Cette méthode prend en paramètre le nom du template et retourne un objet de type `Response` avec pour contenu le contenu de notre template. Voici le contrôleur modifié en conséquence :

Code : PHP

```
<?php  
  
// src/Sdz/BlogBundle/Controller/BlogController.php  
  
namespace Sdz\BlogBundle\Controller;  
  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\HttpFoundation\Response;  
  
class BlogController extends Controller  
{  
    public function indexAction()  
    {  
        return $this->render('SdzBlogBundle:Blog:index.html.twig');  
    }  
}
```

Nous n'avons modifié que la ligne 14. La convention pour le nom du template est la même que pour le nom du contrôleur, souvenez-vous : `:NomDuBundle:NomDuContrôleur:NomDeLAction`.

Maintenant, retournez sur la page http://localhost/Symfony/web/app_dev.php/hello-world et profitez !

Hello World !

Le Hello World est un grand classique en programmation. Il signifie énormément, car cela veut dire que vous avez réussi à exécuter le programme pour accomplir une tâche simple : afficher ce hello world !

Notre vue Hello World s'affiche



Vous avez des problèmes d'accents ? Faites attention à bien définir l'encodage de vos templates en UTF8 sans BOM.



Notez également l'apparition de la `toolbar` en bas de la page. Je vous l'avais déjà dit, Symfony2 l'ajoute automatiquement lorsqu'il détecte la balise fermante `</body>`. C'est pour cela que nous ne l'avions pas tout à l'heure avec notre « Hello World ! » tout simple.

Vous voulez vous amuser un peu avec les variables Twig ? Modifiez la ligne du `return` du contrôleur pour rajouter un deuxième argument à la méthode `render()` :

Code : PHP

```
<?php  
return $this->render('SdzBlogBundle:Blog:index.html.twig',  
array('nom' => 'winzou'));
```

Puis modifiez votre template en remplaçant la balise `<h1>` par la suivante :

Code : HTML

```
<h1>Hello {{ nom }} !</h1>
```

C'est tout ! Rechargez la page. Bonjour à vous également. 😊 On verra plus en détail le passage de variables dans le chapitre dédié à Twig bien évidemment.

Notre objectif : créer un blog

Le fil conducteur : un blog

Tout au long de ce cours, nous construirons un blog. Cela me permet d'utiliser des exemples cohérents entre eux et de vous montrer comment construire un blog de toutes pièces. Bien sûr, libre à vous d'adapter les exemples au projet que vous souhaitez mener, je vous y encourage, même !

Le choix du blog n'est pas très original, mais il permet que l'on se comprenne bien : vous savez déjà ce qu'est un blog, vous comprendrez donc, en théorie, tous les exemples.

Notre blog

Le blog que nous allons créer est très simple. En voici les grandes lignes :

- Nous aurons des articles auxquels nous attacherons des tags.
- Nous pourrons lire, écrire, éditer et rechercher des articles.
- Nous pourrons créer, modifier et supprimer des tags.
- Au début, nous n'aurons pas de système de gestion des utilisateurs : nous devrons saisir notre nom lorsque nous rédigerons un article. Puis nous rajouterons la couche utilisateur.
- Au début, il n'y aura pas de système de commentaires. Puis nous ajouterons cette couche commentaire.

Un peu de nettoyage

Avec tous les éléments générés par Symfony2 et les nôtres, il y a un peu de redondance. Vous pouvez donc supprimer joyeusement :

- Le contrôleur `Controller/DefaultController.php` ;
- Les vues dans le répertoire `Resources/views/Default` ;
- La route `SdzBlogBundle_homepage` dans `Resources/config/routing.yml`.

Ainsi que tout ce qui concerne le bundle `AcmeDemoBundle`, un bundle de démonstration intégré dans la distribution standard de Symfony2 et dont nous ne nous servirons pas. Supprimez donc :

- Le répertoire `src/Acme` ;
- La ligne 26 du fichier `app/AppKernel.php`, celle qui active le bundle `AcmeDemoBundle` (`$bundles[] = new Acme\DemoBundle\AcmeDemoBundle();`) ;
- Les 3 premières routes dans le fichier `app/config/routing_dev.yml` (`_welcome`, `_demo_secured` et `_demo`).

Schéma de développement sous Symfony2

Si vous rafraîchissez la page pour vérifier que tout est bon, il est possible que vous ayez une erreur ! En effet, il faut prendre dès maintenant un réflexe Symfony2 : vider le cache. Car Symfony, pour nous offrir autant de fonctionnalités et être si rapide, utilise beaucoup son cache (des calculs qu'il ne fait qu'une fois puis qu'il stocke). Or après certaines modifications, le cache n'est plus à

jour et il se peut que cela génère des erreurs. Deux cas de figure :

- En mode « prod », c'est simple, Symfony2 ne vide jamais le cache. Cela lui permet de ne faire aucune vérification sur la validité du cache (ce qui prend du temps), et de servir les pages très rapidement à vos visiteurs. La solution : vider le cache à la main *à chaque fois que vous faites des changements*. Cela se fait grâce à la commande `php app/console cache:clear --env=prod`.
- En mode « dev », c'est plus simple et plus compliqué. Lorsque vous modifiez votre code, Symfony reconstruit une bonne partie du cache à la prochaine page que vous chargez. Donc pas forcément besoin de vider le cache. Seulement, comme il ne reconstruit pas tout, il peut parfois apparaître des bugs. Dans ce cas, un petit `php app/console cache:clear` résout le problème en trois secondes !



Parfois, il se peut que la commande `cache:clear` génère des erreurs lors de son exécution. Dans ce cas, essayez de relancer la commande, parfois une deuxième passe peut résoudre les problèmes. Dans le cas contraire, supprimez le cache à la main en supprimant simplement le répertoire `app/cache/dev` (ou `app/cache/prod` suivant l'environnement).

Typiquement, un schéma classique de développement est le suivant :

- Je fais des changements, je teste ;
- Je fais des changements, je teste ;
- Je fais des changements, je teste : ça ne marche pas, je vide le cache : ça marche ;
- Je fais des changements, je teste ;
- Je fais des changements, je teste ;
- Je fais des changements, je teste : ça ne marche pas, je vide le cache : ça marche ;
- ...
- En fin de journée, j'envoie tout sur le serveur de production, je vide obligatoirement le cache, je teste : ça marche.

Évidemment, quand je dis « je teste : ça ne marche pas », j'entends « ça devrait marcher et l'erreur rencontrée est étrange ». Si vous faites une erreur dans votre propre code, ce n'est pas un `cache:clear` qui va la résoudre !

Pour conclure

Et voilà, nous avons créé une page de A à Z ! Voici plusieurs remarques sur ce chapitre.

D'abord, ne vous affolez pas si vous n'avez pas tout compris. Le but de ce chapitre était de vous donner une vision globale d'une page Symfony2. Vous avez des notions de bundles, de routes, de contrôleurs et de templates : vous savez presque tout ! Il ne reste plus qu'à approfondir chacune de ces notions, ce que nous ferons dès le prochain chapitre.

Ensuite, sachez que *tout* n'est pas à refaire lorsque vous créez une deuxième page. Je vous invite là, maintenant, à créer une page `/byebye-world` et voyez si vous y arrivez. Dans le cas contraire, relisez ce chapitre, puis si vous ne trouvez pas votre erreur, n'hésitez pas à poser votre question sur le [forum PHP](#), d'autres Zéros qui sont passés par là seront ravis de vous aider.



Sur le forum, pensez à mettre le tag `[Symfony2]` dans le titre de votre sujet, afin de s'y retrouver.

Enfin, le code source final du blog que nous allons construire ensemble est disponible à l'adresse suivante : www.tutoriel-symfony2.fr/livre/codesource. Il est un peu tôt pour que vous alliez le voir, car il contient le code final alors que nous allons le construire pas à pas grâce à ce cours. Cependant, il peut être d'une bonne aide, allez y jeter un oeil de temps en temps.

Allez, préparez-vous pour la suite, les choses sérieuses commencent !

En résumé

- Le rôle du routeur est de déterminer quel route utiliser pour la requête courante.
- Le rôle d'une route est d'associer une URL à une action du contrôleur.
- Le rôle du contrôleur est de retourner au noyau un objet `Response`, qui contient la réponse HTTP à envoyer à l'internaute (page HTML ou redirection).
- Le rôle des vues est de mettre en forme les données que le contrôleur lui donne, afin de former une page HTML, un flux

RSS, un e-mail, etc.

Le routeur de Symfony2

Comme nous avons pu le voir, le rôle du routeur est, à partir d'une URL, de déterminer quel contrôleur appeler et avec quels arguments. Cela permet de configurer son application pour avoir de très belles URL, ce qui est important pour le référencement et même pour le confort des visiteurs. Soyons d'accord, l'URL `/article/le-système-de-route` est bien plus sexy que `index.php?contrôleur=article&méthode=voir&id=5` !

Vous avez sans doute déjà entendu parler d'*URL Rewriting* ? Le routeur, bien que différent, permet effectivement de faire l'équivalent de l'*URL Rewriting*, mais il le fait côté PHP, et donc est bien mieux intégré à notre code.

Le fonctionnement

L'objectif de ce chapitre est de vous transmettre toutes les connaissances pour pouvoir créer ce que l'on appelle un fichier de *mapping* des routes (un fichier de correspondances, en français). Ce fichier, généralement situé dans `votreBundle/Resources/config/routing.yml`, contient la définition des routes. Chaque route fait la correspondance entre une URL et le contrôleur à appeler. Je vous invite à mettre dès maintenant les routes présentées au code suivant dans le fichier, nous allons travailler dessus dans ce chapitre :

Code : YAML

```
# src/sdz/BlogBundle/Resources/config/routing.yml

sdzblog_accueil:
    path:      /blog
    defaults: { _controller: SdzBlogBundle:Blog:index }

sdzblog_voir:
    path:      /blog/article/{id}
    defaults: { _controller: SdzBlogBundle:Blog:voir }

sdzblog_ajouter:
    path:      /blog/ajouter
    defaults: { _controller: SdzBlogBundle:Blog:ajouter }
```

 Petit rappel au cas où : l'indentation se fait avec 4 espaces par niveau, et non avec des tabulations.

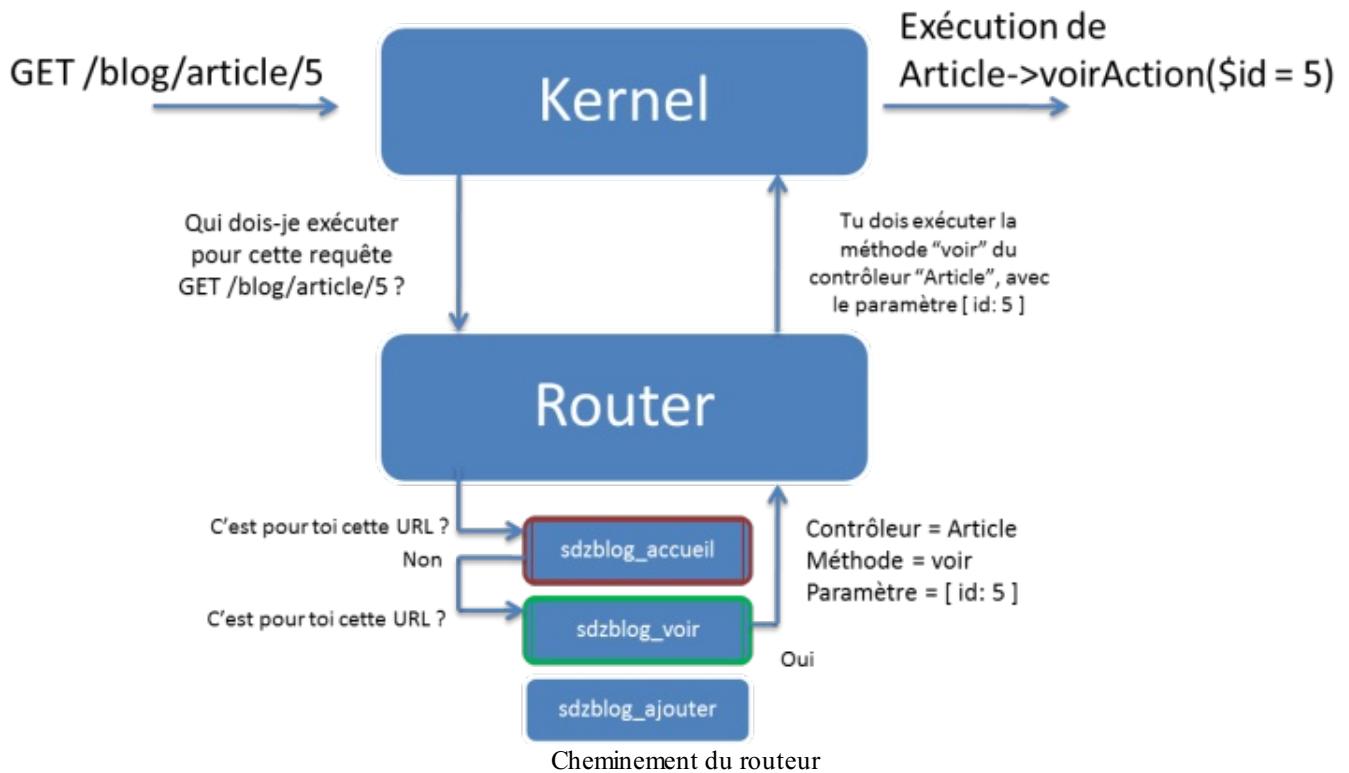
 Vous pouvez supprimer la route `helloTheWorld` que nous avons créée au chapitre précédent, elle ne nous resservira plus. Vous pouvez voir qu'à la place nous avons maintenant une route `sdzblog_accueil`, qui pointe vers la même action du contrôleur. 😊

Fonctionnement du routeur

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

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

Le but du routeur est donc, à partir d'une URL, de trouver la route correspondante et de retourner le contrôleur que veut cette route. Pour trouver la bonne route, le routeur va les parcourir une par une, dans l'ordre du fichier, et s'arrêter à la première route qui fonctionne. La figure suivante est un schéma équivalent au chapitre précédent, mais actualisé pour notre fichier de routes précédent.



Et voici en texte le fonctionnement, pas à pas :

1. On appelle l'URL /blog/article/5.
2. Le routeur essaie de faire correspondre cette URL avec le path de la première route. Ici, /blog/article/5 ne correspond pas du tout à /blog (ligne path de la première route).
3. Le routeur passe donc à la route suivante. Il essaie de faire correspondre /blog/article/5 avec /blog/article/{id}. Nous le verrons plus loin, mais {id} est un paramètre, une sorte de joker « je prends tout ». Cette route correspond, car nous avons bien :
 - /blog/article (URL) = /blog/article (route) ;
 - 5 (URL) = {id} (route).
4. Le routeur s'arrête donc, il a trouvé sa route.
5. Il demande à la route : « Quel contrôleur souhaites-tu appeler, et avec quels paramètres ? », la route répond : « Je veux le contrôleur SdzBlogBundle:Blog:voir, avec le paramètre \$id = 5. »
6. Le routeur renvoie donc ces informations au Kernel (le noyau de Symfony2).
7. Le noyau va exécuter le bon contrôleur !

Dans le cas où le routeur ne trouve aucune route correspondante, le noyau de Symfony2 va déclencher une erreur 404.

Pour chaque page, il est possible de visualiser toutes les routes que le routeur essaie une à une, et celle qu'il utilise finalement. C'est le Profiler qui s'occupe de tracer cela, accessible depuis la barre d'outils : cliquez sur le nom de la route dans la barre d'outils, « sdzblog_accueil » si vous êtes sur la page /blog. Ce lien vous amène dans l'onglet « Request » du Profiler, mais allez dans l'onglet « Routing » qui nous intéresse. Vous devriez obtenir la figure suivante.

View all Profile for: GET http://localhost/Symfony/web/app_dev.php/blog by 127.0.0.1 at Wed, 11 Jul 2012 22:13:52 +0200		
Routing for "/blog"		
Route: sdzblog_accueil		
Route parameters: No parameters		
Route matching logs		
Route name	Pattern	Log
_wdt	/_wdt/{token}	Pattern "/_wdt/{token}" does not match
_profiler_search	/_profiler/search	Pattern "/_profiler/search" does not match
_profiler_purge	/_profiler/purge	Pattern "/_profiler/purge" does not match
_profiler_info	/_profiler/info/{about}	Pattern "/_profiler/info/{about}" does not match
_profiler_import	/_profiler/import	Pattern "/_profiler/import" does not match
_profiler_export	/_profiler/export/{token}.txt	Pattern "/_profiler/export/{token}.txt" does not match
_profiler_phpinfo	/_profiler/phpinfo	Pattern "/_profiler/phpinfo" does not match
_profiler_search_results	/_profiler/{token}/search/results	Pattern "/_profiler/{token}/search/results" does not match
_profiler	/_profiler/{token}	Pattern "/_profiler/{token}" does not match
_profiler_redirect	/_profiler/	Pattern "/_profiler/" does not match
_configurator_home	/_configurator/	Pattern "/_configurator/" does not match
_configurator_step	/_configurator/step/{index}	Pattern "/_configurator/step/{index}" does not match
_configurator_final	/_configurator/final	Pattern "/_configurator/final" does not match
sdzblog_accueil	/blog	Route matches!

Liste des routes enregistrées par le routeur



Vous pouvez voir qu'il y a déjà pas mal de routes définies alors que nous n'avons rien fait. Ces routes qui commencent par /_profiler/ sont les routes nécessaires au Profiler, dans lequel vous êtes. Eh oui, c'est un bundle également, le bundle WebProfilerBundle !

Convention pour le nom du contrôleur

Vous l'avez vu, lorsque l'on définit le contrôleur à appeler dans la route, il y a une convention à respecter : la même que pour appeler un template (nous l'avons vue au chapitre précédent). Un rappel ne fait pas de mal : lorsque vous écrivez « SdzBlogBundle:Blog:voir », vous avez trois informations :

- « SdzBlogBundle » est le nom du bundle dans lequel aller chercher le contrôleur. En terme de fichier, cela signifie pour Symfony2 : « Va voir dans le répertoire de ce bundle. ». Dans notre cas, Symfony2 ira voir dans src/Sdz/BlogBundle.
- « Blog » est le nom du contrôleur à ouvrir. En terme de fichier, cela correspond à controller/BlogController.php dans le répertoire du bundle. Dans notre cas, nous avons comme chemin absolu src/Sdz/BlogBundle/controller/BlogController.php.
- « voir » est le nom de l'action à exécuter au sein du contrôleur. Attention, lorsque vous définissez cette méthode dans le contrôleur, vous devez la faire suivre du suffixe « Action », comme ceci : <?php **public function voirAction ()**.

Les routes de base Créer une route

Étudions la première route plus en détail :

Code : YAML

```
# src/Sdz/BlogBundle/config/Resources/routing.yml
sdzblog_accueil:
```

```
path:      /blog
defaults: { _controller: SdzBlogBundle:Blog:index }
```

Ce bloc représente ce que l'on nomme une « route ». Elle est constituée au minimum de trois éléments :

- `sdzblog_accueil` est le nom de la route. Il n'a aucune importance dans le travail du routeur, mais il interviendra lorsque l'on voudra générer des URL : eh oui, on n'écrira pas l'URL à la main, mais on fera appel au routeur pour qu'il fasse le travail à notre place ! Retenez donc pour l'instant qu'il faut qu'un nom soit unique et clair. On a donc préfixé les routes de « `sdzblog` » pour l'unicité entre bundles.
- `path: /blog` est l'URL sur laquelle la route s'applique. Ici, « `/blog` » correspond à une URL absolue du type `http://www.monsite.com/blog`.
- `defaults: { _controller: SdzBlogBundle:Blog:index }` correspond au contrôleur à appeler.

Vous avez maintenant les bases pour créer une route simple !

Créer une route avec des paramètres

Reprendons la deuxième route de notre exemple :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_voir:
    path:      /blog/article/{id}
    defaults: { _controller: SdzBlogBundle:Blog:voir }
```

Grâce au paramètre `{id}` dans le `path` de notre route, toutes les URL du type `/blog/article/*` seront gérées par cette route, par exemple : `/blog/article/5` ou `/blog/article/654`, ou même `/blog/article/sodfihsodfh` (on n'a pas encore dit que `{id}` devait être un nombre, patience !). Par contre, l'URL `/blog/article` ne sera pas interceptée, car le paramètre `{id}` n'est pas renseigné. En effet, les paramètres sont par défaut obligatoires, nous verrons quand et comment les rendre facultatifs plus loin dans ce chapitre.

Mais si le routeur s'arrêtait là, il n'aurait aucun intérêt. Toute sa puissance réside dans le fait que ce paramètre `{id}` est accessible depuis votre contrôleur ! Si vous appelez l'URL `/blog/article/5`, alors depuis votre contrôleur vous aurez la variable `$id` (du nom du paramètre) qui aura pour valeur « `5` ». Je vous invite à créer la méthode correspondante dans le contrôleur :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    // ... ici la méthode indexAction() que l'on a déjà créée

    // La route fait appel à SdzBlogBundle:Blog:voir, on doit donc
    // définir la méthode voirAction
    // On donne à cette méthode l'argument $id, pour correspondre au
    // paramètre {id} de la route
    public function voirAction($id)
    {
        // $id vaut 5 si l'on a appellé l'URL /blog/article/5
```

```

    // Ici, on récupèrera depuis la base de données l'article
    // correspondant à l'id $id
    // Puis on passera l'article à la vue pour qu'elle puisse
    // l'afficher

    return new Response("Affichage de l'article d'id : ".$id.".");
}
}

```

N'oubliez pas de tester votre code à l'adresse suivante :

http://localhost/Symfony/web/app_dev.php/blog/article/5, et amusez-vous à changer la valeur du paramètre.

Vous pouvez bien sûr multiplier les paramètres au sein d'une même route. Ajoutez cette route juste après la route sdzblob_voir, pour l'exemple :

Code : YAML

```

# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblob_voir_slug:
    path:      /blog/{annee}/{slug}.{format}
    defaults: { _controller: SdzBlogBundle:Blog:voirSlug }

```

Cette route permet d'intercepter les URL suivantes : /blog/2011/mon-weekend.html ou /blog/2012/symfony.xml, etc. Et voici la méthode correspondante qu'on aurait côté contrôleur :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    // ... ici les méthodes indexAction() et voirAction() que l'on a
    // déjà créées

    // On récupère tous les paramètres en arguments de la méthode
    public function voirSlugAction($slug, $annee, $format)
    {
        // Ici le contenu de la méthode
        return new Response("On pourrait afficher l'article
correspondant au slug '".$slug."', créé en ".$annee." et au format
".$format.".");
    }
}

```

 Notez que l'ordre des arguments dans la définition de la méthode voirSlugAction() n'a pas d'importance. La route fait la correspondance à partir du nom des variables utilisées, non à partir de leur ordre. C'est toujours bon à savoir !

Revenez à notre route et notez également le point entre les paramètres {slug} et {format} : vous pouvez en effet séparer

vos paramètres soit avec le slash (« / »), soit avec le point (« . »). Veillez donc à ne pas utiliser de point dans le contenu de vos paramètres. Par exemple, pour notre paramètre {slug}, une URL /blog/2011/mon-weekend.etait.bien.html ne va pas correspondre à cette route, car :

- {annee} = 2011 ;
- {slug} = mon-weekend ;
- {format} = etait ;
- ? = bien ;
- ? = html ;

La route attend des paramètres à mettre en face de ces dernières valeurs, et comme il n'y en a pas cette route dit : « Cette URL ne me correspond pas, passez à la route suivante. » Attention donc à ce petit détail. 😊

Les routes avancées

Créer une route avec des paramètres et leurs contraintes

Nous avons créé une route avec des paramètres, très bien. Mais si quelqu'un essaie d'atteindre l'URL /blog/oaisd/aouish.oasidh, eh bien, rien ne l'en empêche ! Et pourtant, « oaisd » n'est pas tellement une année valide ! La solution ? Les contraintes sur les paramètres. Reprenons notre dernière route sdzblob_voir_slug :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
sdzblob_voir_slug:
    path:    /blog/{annee}/{slug}.{format}
    defaults: { _controller: SdzBlogBundle:Blog:voirSlug }
```

Nous voulons ne récupérer que les bonnes URL où l'année vaut « 2010 » et non « oshidf », par exemple. Cette dernière devrait retourner une erreur 404 (page introuvable). Pour cela, il nous suffit qu'aucune route ne l'intercepte ; ainsi, le routeur arrivera à la fin du fichier sans aucune route correspondante et il déclenchera tout seul une erreur 404.

Comment faire pour que notre paramètre {année} n'intercepte pas « oshidf » ? C'est très simple :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
sdzblob_voir_slug:
    path:    /blog/{annee}/{slug}.{format}
    defaults: { _controller: SdzBlogBundle:Blog:voirSlug }
    requirements:
        annee: \d{4}
        format: html|xml
```

Nous avons ajouté la section `requirements`. Comme vous pouvez le voir, on utilise les expressions régulières pour déterminer les contraintes que doivent respecter les paramètres. Ici :

- `\d{4}` veut dire « quatre chiffres à la suite ». L'URL /blog/sdff/mon-weekend.html ne sera donc pas interceptée. Vous l'avez reconnue, c'est une expression régulière. Vous pouvez utiliser n'importe laquelle, je vous invite à lire [le cours correspondant de M@teo21](#).
- `html|xml` signifie « soit HTML, soit XML ». L'URL /blog/2011/mon-weekend.rss ne sera donc pas interceptée.



N'hésitez surtout pas à faire les tests ! Cette route est opérationnelle, nous avons créé l'action correspondante dans le contrôleur. Essayez donc de bien comprendre quels paramètres sont valides, lesquels ne le sont pas. Vous pouvez également changer la section `requirements`.

Maintenant, nous souhaitons aller plus loin. En effet, si le « .xml » est utile pour récupérer l'article au format XML (pourquoi pas ?), le « .html » semble inutile : par défaut, le visiteur veut toujours du HTML. Il faut donc rendre le paramètre `{format}` facultatif.

Utiliser des paramètres facultatifs

Reprendons notre route et ajoutons-y la possibilité pour `{format}` de ne pas être renseigné :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_voir_slug:
    path:      /blog/{annee}/{slug}.{format}
    defaults: { _controller: SdzBlogBundle:Blog:voirSlug, format: html }
    requirements:
        annee: \d{4}
        format: html|xml
```

Nous avons juste ajouté une valeur par défaut dans le tableau `defaults: format: html`. C'est aussi simple que cela ! Ainsi, l'URL `/blog/2011/mon-weekend` sera bien interceptée et le paramètre `format` sera mis à sa valeur par défaut, à savoir « `html` ». Au niveau du contrôleur, rien ne change : vous gardez l'argument `$format` comme avant et celui-ci vaudra « `html` », la valeur par défaut.

Utiliser des « paramètres système »

Prenons l'exemple de notre paramètre `{format}` : lorsqu'il vaut « `xml` », vous allez afficher du XML et devrez donc envoyer le header avec le bon Content-type. Les développeurs de Symfony2 ont pensé à nous et prévu des « paramètres système ». Ils s'utilisent exactement comme des paramètres classiques, mais effectuent automatiquement des actions supplémentaires :

- Le paramètre `{_format}` : lorsqu'il est utilisé (comme notre paramètre `{format}`, ajoutez juste un *underscore*), alors un header avec le Content-type correspondant est envoyé. Exemple : vous appelez `/blog/2011/mon-weekend.xml` et le routeur va dire à l'objet Request que l'utilisateur demande du XML. Ainsi, l'objet Response enverra un header Content-type: `application/xml`. Vous n'avez plus à vous en soucier ! Depuis le contrôleur, vous pouvez récupérer ce format soit avec l'argument `$_format` comme n'importe quel autre argument, soit via la méthode `getRequestFormat()` de l'objet Request. Par exemple : `<?php $this->get('request')->getRequestFormat()`.
- Le paramètre `{_locale}` : lorsqu'il est utilisé, il va définir la langue dans laquelle l'utilisateur souhaite obtenir la page. Ainsi, si vous avez défini des fichiers de traduction ou si vous employez des bundles qui en utilisent, alors les traductions dans la langue du paramètre `{_locale}` seront chargées. Pensez à mettre un `requirements:` sur la valeur de ce paramètre pour éviter que vos utilisateurs ne demandent le russe alors que votre site n'est que bilingue français-anglais.

Ajouter un préfixe lors de l'import de nos routes

Vous avez remarqué que nous avons mis `/blog` au début du path de chacune de nos routes. En effet, on crée un blog, on aimerait donc que toutes les URL aient ce préfixe `/blog`. Au lieu de les répéter à chaque fois, Symfony2 vous propose de rajouter un préfixe lors de l'import du fichier de notre bundle.

Modifiez donc le fichier `app/config/routing.yml` comme suit :

Code : Autre

```
# app/config/routing.yml
```

```
SdzBlogBundle:  
    resource:  "@SdzBlogBundle/Resources/config/routing.yml"  
    prefix:   /blog
```

Vous pouvez ainsi enlever la partie /blog de chacune de vos routes. Bonus : si un jour vous souhaitez changer /blog par /blogdemichel, vous n'aurez qu'à modifier une seule ligne. 😊

Générer des URL

Pourquoi générer des URL ?

J'ai mentionné précédemment que le routeur pouvait aussi générer des URL à partir du nom des routes. En effet, vu que le routeur a toutes les routes à sa disposition, il est capable d'associer une route à une certaine URL, mais également de reconstruire l'URL correspondant à une certaine route. Ce n'est pas une fonctionnalité annexe, mais bien un outil puissant que nous avons là !

Par exemple, nous avons une route nommée « sdzblog_voir » qui écoute l'URL /blog/article/{id}. Vous décidez un jour de raccourcir vos URL et vous aimeriez bien que vos articles soient disponibles depuis /blog/a/{id}. Si vous aviez écrit toutes vos URL à la main, vous auriez dû toutes les changer à la main, une par une. Grâce à la génération d'URL, vous ne modifiez que la route : ainsi, toutes les URL générées seront mises à jour ! C'est un exemple simple, mais vous pouvez trouver des cas bien réels et tout aussi gênants sans la génération d'URL.

Comment générer des URL ?

1. Depuis le contrôleur

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

Depuis un contrôleur, c'est la méthode <?php \$this->generateUrl()> qu'il faut appeler. Par exemple :

Code : PHP

```
<?php  
// src/Sdz/BlogBundle/Controller/BlogController.php  
  
namespace Sdz\BlogBundle\Controller;  
  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\HttpFoundation\Response;  
  
class BlogController extends Controller  
{  
    public function indexAction()  
    {  
        // On fixe un id au hasard ici, il sera dynamique par la suite,  
        // évidemment  
        $id = 5;  
  
        // On veut avoir l'URL de l'article d'id $id.  
        $url = $this->generateUrl('sdzblog_voir', array('id' => $id));  
        // $url vaut « /blog/article/5 »  
  
        // On redirige vers cette URL (ça ne sert à rien, on est  
        // d'accord, c'est pour l'exemple !)  
        return $this->redirect($url);  
    }  
}
```

Pour générer une URL absolue, lorsque vous l'envoyez par e-mail, par exemple, il faut mettre le troisième argument à true.

Exemple :

Code : PHP

```
<?php
$url = $this->generateUrl('sdzblog_voir', array('id' => $id), true);
```

Ainsi, \$url vaut `http://monsite.com/blog/article/5` et pas uniquement `/blog/article/5`.

2. Depuis une vue Twig

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

Code : HTML & Django

```
{# Dans une vue Twig, en considérant bien sûr que la variable
article_id est disponible #-}

<a href="{{ path('sdzblog_voir', { 'id': article_id }) }}>Lien vers
l'article d'id {{ article_id }}</a>
```

Et pour générer une URL absolue depuis Twig, pas de troisième argument, mais on utilise la fonction `url()` au lieu de `path()`. Elle s'utilise exactement de la même manière, seul le nom change.

Voilà : vous savez générer des URL, ce n'était vraiment pas compliqué. Pensez bien à utiliser la fonction `{{ path }}` pour tous vos liens dans vos templates. 😊

Application : les routes de notre blog

Construction des routes

Revenons à notre blog. Maintenant que nous savons créer des routes, je vous propose de faire un premier jet de ce que seront nos URL. Voici les routes que je vous propose de créer, libre à vous d'en changer.

Page d'accueil

On souhaite avoir une URL très simple pour la page d'accueil : `/blog`. Comme `/blog` est défini comme préfixe lors du chargement des routes de notre bundle, le path ici est « / ». Mais on veut aussi pouvoir parcourir les articles plus anciens, donc il nous faut une notion de page courante. En ajoutant le paramètre facultatif `{page}`, nous aurons :

<code>/blog</code>	<code>page = 1</code>
<code>/blog/1</code>	<code>page = 1</code>
<code>/blog/2</code>	<code>page = 2</code>

C'est plutôt joli, non ? Voici la route :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_accueil:
    path:      '/{page}'
    defaults:  { _controller: SdzBlogBundle:Blog:index, page: 1 }
    requirements:
```

```
page: \d*
```

Page de visualisation d'un article

Pour la page d'un unique article, la route est très simple. Il suffit juste de bien mettre un paramètre `{id}` qui nous servira à récupérer le bon article côté contrôleur. Voici la route :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_voir:
    path: /article/{id}
    defaults: { _controller: SdzBlogBundle:Blog:voir }
```

Ajout, modification et suppression

Les routes sont simples :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_ajouter:
    path: /ajouter
    defaults: { _controller: SdzBlogBundle:Blog:ajouter }

sdzblog_modifier:
    path: /modifier/{id}
    defaults: { _controller: SdzBlogBundle:Blog:modifier }
    requirements:
        id: \d+

sdzblog_supprimer:
    path: /supprimer/{id}
    defaults: { _controller: SdzBlogBundle:Blog:supprimer }
    requirements:
        id: \d+
```

Récapitulatif

Voici le code complet de notre fichier `src/Sdz/BlogBundle/Resources/config/routing.yml` :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_accueil:
    path: /{page}
    defaults: { _controller: SdzBlogBundle:Blog:index, page: 1 }
    requirements:
        page: \d*

sdzblog_voir:
    path: /article/{id}
```

```
defaults: { _controller: SdzBlogBundle:Blog:voir }
requirements:
    id: \d+

sdzblog_ajouter:
    path:      /ajouter
    defaults: { _controller: SdzBlogBundle:Blog:ajouter }

sdzblog_modifier:
    path:      /modifier/{id}
    defaults: { _controller: SdzBlogBundle:Blog:modifier }
    requirements:
        id: \d+

sdzblog_supprimer:
    path:      /supprimer/{id}
    defaults: { _controller: SdzBlogBundle:Blog:supprimer }
    requirements:
        id: \d+
```

N'oubliez pas de bien ajouter le préfixe `/blog` lors de l'import de ce fichier, dans `app/config/routing.yml`:

Code : YAML

```
# app/config/routing.yml

SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix:  /blog
```

Pour conclure

Ce chapitre est terminé, et vous savez maintenant tout ce qu'il faut savoir sur le routeur et les routes.

Retenez que ce système de routes vous permet premièrement d'avoir des belles URL, et deuxièmement de découpler le nom de vos URL du nom de vos contrôleurs. Ajoutez à cela la génération d'URL, et vous avez un système extrêmement flexible et maintenable.

Le tout sans trop d'efforts !

Pour plus d'informations sur le système de routes, n'hésitez pas à lire la [documentation officielle](#).

En résumé

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

Les contrôleurs avec Symfony2

Ah, le contrôleur ! Vous le savez, c'est lui qui contient toute la logique de notre site internet. Cependant, cela ne veut pas dire qu'il contient beaucoup de code. En fait, il ne fait qu'utiliser des services, les modèles et appeler la vue. Finalement, c'est un chef d'orchestre qui se contente de faire la liaison entre tout le monde.

Nous verrons dans ce chapitre ses droits, mais aussi son devoir ultime : retourner une réponse !

Le rôle du contrôleur

Retourner une réponse

Je vous l'ai dit de nombreuses fois depuis le début de ce cours : le rôle du contrôleur est de retourner une réponse.



Mais concrètement, qu'est-ce que cela signifie, « retourner une réponse » ?

Souvenez-vous, Symfony2 s'est inspiré des concepts du protocole HTTP. Il existe dans Symfony2 une classe Response. Retourner une réponse signifie donc tout simplement : instancier un objet Response, disons \$response, et faire un return \$response.

Voici le contrôleur le plus simple qui soit, c'est le contrôleur qu'on avait à créer dans un des chapitres précédents. Il dispose d'une seule méthode, nommée « index », et retourne une réponse qui ne contient que « Hello World ! » :

Code : PHP

```
<?php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function indexAction()
    {
        return new Response("Hello World !");
    }
}
```

Et voilà, votre contrôleur remplit parfaitement son rôle !

Bien sûr, vous n'irez pas très loin en sachant juste cela. C'est pourquoi la suite de ce chapitre est découpée en deux parties :

- Les objets Request et Response qui vont vous permettre de construire une réponse en fonction de la requête ;
- Les services qui vont vous permettre de réaliser tout le travail nécessaire pour préparer le contenu de votre réponse.

Manipuler l'objet Request

Les paramètres de la requête

Heureusement, toutes les requêtes que l'on peut faire sur un site internet ne sont pas aussi simples que notre « Hello World ! ». Dans bien des cas, une requête contient des paramètres : l'id d'un article à afficher, le nom d'un membre à chercher dans la base de données, etc. Les paramètres sont la base de toute requête : la construction de la page à afficher dépend de chacun des paramètres en entrée.

Ces paramètres, nous savons déjà les gérer, nous l'avons vu dans le chapitre sur le routeur. Mais voici un petit rappel.

Les paramètres contenus dans les routes

Tout d'abord côté route, souvenez-vous, on utilisait déjà des paramètres. Prenons l'exemple de la route sdzblog_voir :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_voir:
    path:      /article/{id}
    defaults: { _controller: SdzBlogBundle:Blog:voir }
    requirements:
        id: \d+
```

Ici, le paramètre `{id}` de la requête est récupéré par la route, qui va le transformer en argument `$id` pour le contrôleur. On a déjà fait la méthode correspondante dans le contrôleur, la voici pour rappel :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    // ...

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

Voici donc la première manière de récupérer des arguments : ceux contenus dans la route.

Les paramètres hors routes

En plus des paramètres de routes que nous venons de voir, vous pouvez récupérer les autres paramètres de l'URL, disons, « à l'ancienne ». Prenons par exemple l'URL `/blog/article/5?tag=vacances`, il nous faut bien un moyen pour récupérer ce paramètre `tag` ! C'est ici qu'intervient l'objet `Request`. Tout d'abord, voici comment récupérer la requête depuis un contrôleur :

Code : PHP

```
<?php
$request = $this->getRequest();
```

Voilà, c'est aussi simple que cela ! Maintenant que nous avons notre requête, récupérons nos paramètres :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```

use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    // ...

    public function voirAction($id)
    {
        // On récupère la requête
        $request = $this->getRequest();

        // On récupère notre paramètre tag
        $tag = $request->query->get('tag');

        return new Response("Affichage de l'article d'id : ".$id.", avec
le tag : ".$tag);
    }
}

```

Et vous n'avez plus qu'à tester le résultat : </blog/article/9?tag=vacances>.

Nous avons utilisé `<?php $request->query` pour récupérer les paramètres de l'URL passés en GET, mais vous savez qu'il existe d'autres types de paramètres :

Type de paramètres	Méthode Symfony2	Méthode traditionnelle	Exemple
Variables d'URL	<code><?php \$request->query</code>	<code>\$_GET</code>	<code><?php \$request->query->get('tag')</code>
Variables de formulaire	<code><?php \$request->request</code>	<code>\$_POST</code>	<code><?php \$request->request->get('tag')</code>
Variables de cookie	<code><?php \$request->cookies</code>	<code>\$_COOKIE</code>	<code><?php \$request->cookies->get('tag')</code>
Variables de serveur	<code><?php \$request->server</code>	<code>\$_SERVER</code>	<code><?php \$request->server->get('REQUEST_URI')</code>
Variables d'entête	<code><?php \$request->headers</code>	<code>\$_SERVER['HTTP_*']</code>	<code><?php \$request->headers->get('USER_AGENT')</code>
Paramètres de route	<code><?php \$request->attributes</code>	n/a	<code><?php \$request->attributes->get('id')</code> Est équivalent à : <code><?php \$id</code>

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

Les autres méthodes de l'objet Request

Heureusement, l'objet Request ne se limite pas à la récupération de paramètres. Il permet de savoir plusieurs choses intéressantes à propos de la requête en cours, voyons ses possibilités.

Récupérer la méthode de la requête HTTP

Pour savoir si la page a été récupérée via GET (clic sur un lien) ou via POST (envoi d'un formulaire), il existe la méthode `<?php $request->getMethod() ?>` :

Code : PHP

```
<?php
if( $request->getMethod() == 'POST' )
{
    // Un formulaire a été envoyé, on peut le traiter ici
}
```

Savoir si la requête est une requête AJAX

Lorsque vous utiliserez AJAX dans votre site, vous aurez sans doute besoin de savoir, depuis le contrôleur, si la requête en cours est une requête AJAX ou non. Par exemple, pour renvoyer du XML ou du JSON à la place du HTML. Pour cela, rien de plus simple !

Code : PHP

```
<?php
if( $request->isXmlHttpRequest() )
{
    // C'est une requête AJAX, retournons du JSON, par exemple
}
```

Toutes les autres

Pour avoir la liste exhaustive des méthodes disponibles sur l'objet Request, je vous invite à lire l'API de cet objet sur [le site de Symfony2](#). Vous y trouverez toutes les méthodes, même si nous avons déjà survolé les principales.

Manipuler l'objet Response**Décomposition de la construction d'un objet Response**

Pour que vous compreniez ce qu'il se passe en coulisses lors de la création d'une réponse, voyons la manière longue et décomposée de construire et de retourner une réponse. Pour l'exemple, traitons le cas d'une page d'erreur 404 (page introuvable) :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    // On modifie voirAction, car elle existe déjà
    public function voirAction($id)
    {
        // On crée la réponse sans lui donner de contenu pour le moment
        $response = new Response();

        // On définit le contenu
        $response->setContent('Ceci est une page d\'erreur 404');

        // On définit le code HTTP
        // Rappelez-vous, 404 correspond à « page introuvable »
        $response->setStatusCode(404);

        // On retourne la réponse
    }
}
```

```
        return $response;
    }
}
```

N'hésitez pas à tester cette page, l'URL est http://localhost/Symfony/web/app_dev.php/blog/article/5 si vous avez gardé les mêmes routes depuis le début.

Je ne vous le cache pas : nous n'utiliserons jamais cette longue méthode ! Lisez plutôt la suite.

Réponses et vues

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

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

// Nous n'avons plus besoin du use pour l'objet Response
// use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // On utilise le raccourci : il crée un objet Response
        // Et lui donne comme contenu le contenu du template
        return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
            'id' => $id,
        ));
    }
}
```

Et voilà, en une seule ligne, c'est bouclé ! C'est comme cela que nous générerons la plupart de nos réponses. Finalement, l'objet Response est utilisé en coulisses, nous n'avons pas à le manipuler directement.

N'oubliez pas de créer la vue associée bien entendu :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/view/Blog/voir.html.twig #-}

<!DOCTYPE html>
<html>
    <head>
        <title>Lecture de l'article {{ id }}</title>
    </head>
    <body>
        <h1>Hello Article n°{{ id }} !</h1>
    </body>
</html>
```

Si vous ne deviez retenir qu'une seule chose de cette section, c'est bien cette méthode `<?php $this->render()`, car c'est vraiment ce que nous utiliserons en permanence. 😊

Réponse et redirection

Vous serez sûrement amenés à faire une redirection vers une autre page. Or notre contrôleur est *obligé* de retourner une réponse. Comment gérer une redirection ? Eh bien, vous avez peut-être évité le piège, mais *une redirection est une réponse HTTP*. Pour faire cela, il existe également un raccourci du contrôleur : la méthode `<?php $this->redirect()`. Elle prend en paramètre l'URL vers laquelle vous souhaitez faire la redirection et s'occupe de créer une réponse, puis d'y définir un header qui contiendra votre URL. En action, cela donne le code suivant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // On utilise la méthode « generateUrl() » pour obtenir l'URL
        // de la liste des articles à la page 2
        // Par exemple
        return $this->redirect( $this->generateUrl('sdzblog_accueil',
array('page' => 2)) );
    }
}
```

Essayez d'aller à l'adresse http://localhost/Symfony/web/app_dev.php/blog/article/5 et vous serez redirigés vers l'accueil !

Changer le Content-type de la réponse

Lorsque vous retournez autre chose que du HTML, il faut que vous changez le Content-type de la réponse. Ce Content-type permet au navigateur qui recevra votre réponse de savoir à quoi s'attendre dans le contenu. Prenons l'exemple suivant : vous recevez une requête AJAX et souhaitez retourner un tableau en JSON :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // Créons nous-mêmes la réponse en JSON, grâce à la fonction
        json_encode()
        $response = new Response(json_encode(array('id' => $id)));
        // Ici, nous définissons le Content-type pour dire que l'on
        // renvoie du JSON
        $response->headers->set('Content-Type', 'application/json');
        return $response;
    }
}
```

```
renvoie du JSON et non du HTML
$response->headers->set('Content-Type', 'application/json');

return $response;

// Nous n'avons pas utilisé notre template ici, car il n'y en a
pas vraiment besoin
}
}
```

Testez le rendu en allant sur http://localhost/Symfony/web/app_dev.php/blog/article/5.

Les différents services

Qu'est-ce qu'un service ?

Je vous en ai déjà brièvement parlé : un service est un script qui remplit un rôle précis et que l'on peut utiliser depuis notre contrôleur.

Imaginez par exemple un service qui a pour but d'envoyer des e-mails. Depuis notre contrôleur, on appelle ce service, on lui donne les informations nécessaires (contenu de l'e-mail, destinataire, etc.), puis on lui dit d'envoyer l'e-mail. Ainsi, toute la logique « création et envoi d'e-mail » se trouve dans ce service et non dans notre contrôleur. Cela nous permet de réutiliser ce service très facilement ! En effet, si vous codez en dur l'envoi d'e-mail dans un contrôleur A et que, plus tard, vous avez envie d'envoyer un autre e-mail depuis un contrôleur B, comment réutiliser ce que vous avez déjà fait ? C'est impossible et c'est exactement pour cela que les services existent.

 Il existe un chapitre sur les services plus loin dans ce tutoriel. N'allez pas le lire maintenant, car il demande des notions que vous n'avez pas encore. Patience, sachez juste que c'est un point incontournable de Symfony2 et que nous le traiterons bien plus en détail par la suite. 

Accéder aux services

Pour accéder aux services depuis votre contrôleur, il faut utiliser la méthode `<?php $this->get()` du contrôleur. Par exemple, le service pour envoyer des e-mails se nomme justement « mailer ». Pour employer ce service, nous faisons donc :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // Récupération du service
        $mailer = $this->get('mailer');

        // Création de l'e-mail : le service mailer utilise
        // SwiftMailer, donc nous créons une instance de Swift_Message
        $message = \Swift_Message::newInstance()
            ->setSubject('Hello zéro !')
            ->setFrom('tutorial@symfony2.com')
            ->setTo('votre@email.com')
            ->setBody('Coucou, voici un email que vous venez de recevoir
        !');

        // Retour au service mailer, nous utilisons sa méthode « send()
    }
}
```

```
>> pour envoyer notre $message  
    $mailer->send($message);  
  
    // N'oublions pas de retourner une réponse, par exemple une  
    page qui afficherait « L'e-mail a bien été envoyé »  
    return new Response('Email bien envoyé');  
}  
}
```

Pour que l'envoi d'e-mail fonctionne, n'oubliez pas de configurer vos paramètres si ce n'est pas déjà fait. Ouvrez le fichier `app/config/parameters.yml` pour modifier les paramètres `mailer_*`. Si vous voulez utiliser votre compte Gmail :

Code : YAML

```
# app/config/parameters.yml  
  
mailer_transport:  gmail  
mailer_host:  
mailer_user:      vous@gmail.com  
mailer_password:  mdp
```

Et si vous voulez utiliser un serveur SMTP classique :

Code : YAML

```
# app/config/parameters.yml  
  
mailer_transport:  smtp  
mailer_host:       smtp.votre-serveur.fr  
mailer_user:       identifiant  
mailer_password:  mdp
```

Chargez la page `/blog/article/5`, et allez lire votre e-mail !



Retenez donc la méthode `<?php $this->get('nom_du_service') !` !

Brève liste des services

Maintenant que vous savez récupérer des services, encore faut-il connaître leur nom ! Et savoir les utiliser ! Ci-après est dressée une courte liste de quelques services utiles.

Templating

Templating est un service qui vous permet de gérer vos templates (vos vues, vous l'aurez compris). En fait, vous avez déjà utilisé ce service... via le raccourci `<?php $this->render !` Voici la version longue d'un `<?php $this->render('MonTemplate')` :

Code : PHP

```
<?php  
// ...  
public function voirAction($id)  
{
```

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

// On récupère le contenu de notre template
$content = $templating-
>render('SdzBlogBundle:Blog:voir.html.twig');

// On crée une réponse avec ce contenu et on la retourne
return new Response($content);
}
```

Le service Templating est utile, par exemple, pour notre e-mail de tout à l'heure. Nous avons écrit le contenu de l'e-mail en dur, ce qui n'est pas bien, évidemment. Nous devrions avoir un template pour cela. Et pour en récupérer le contenu, nous utilisons <?php \$templating->render(). 😊

Une autre fonction de ce service qui peut servir, c'est <?php \$templating->exists('SdzBlogBundle:Blog:inexistant') qui permet de vérifier si « SdzBlogBundle:Blog:inexistant » existe ou non.

Request

Eh oui, encore elle. C'est également un service ! Tout à l'heure on l'a récupéré via \$this->getRequest(), mais on aurait également pu la récupérer de cette façon : <?php \$this->get('request'). En fait, la première méthode n'est qu'un raccourci vers la deuxième.

Session

Les outils de session sont également intégrés dans un service. Vous pouvez le récupérer via <?php \$this->get('session'). Pour définir et récupérer des variables en session, il faut utiliser les méthodes get() et set(), tout simplement :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // Récupération du service
        $session = $this->get('session');

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

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

        // On n'oublie pas de renvoyer une réponse
        return new Response('Désolé je suis une page de test, je n\'ai rien à dire');
    }
}
```

 Pour connaître les variables de la session courante, allez dans le Profiler (la barre d'outils en bas), rubrique Request, puis descendez tout en bas au paragraphe « Session Attributes ». Très utile pour savoir si vous avez bien les variables de session que vous attendez. 

La session se lance automatiquement dès que vous vous en servez. Voyez par exemple à la figure suivante ce que le Profiler me dit sur une page où je n'utilise pas la session.

Session Metadata

Key	Value
Created	'Thu, 01 Jan 70 01:00:00 +0100'
Last used	'Thu, 01 Jan 70 01:00:00 +0100'
Lifetime	0

Session Attributes

No session attributes

constate qu'il n'y a pas d'attribut dans la session

Et voici le Profiler après que nous avons défini la variable `user_id` en session, à la figure suivante.

Session Metadata

Key	Value
Created	'Wed, 11 Jul 12 18:54:35 +0200'
Last used	'Thu, 12 Jul 12 21:46:34 +0200'
Lifetime	'0'

Session Attributes

Key	Value
<code>user_id</code>	91

Ici, on constate que l'attribut `user_id` est bien défini, avec comme valeur 91

Le Profiler nous donne même les informations sur la date de création de la session, etc.

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

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function voirAction($id)
    {
        return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
            'id' => $id
        ));
    }

    // Ajoutez cette méthode ajouterAction :
    public function ajouterAction()
    {
        // Bien sûr, cette méthode devra réellement ajouter l'article
        // Mais faisons comme si c'était le cas
        $this->get('session')->setFlashBag()->add('info', 'Article bien
enregistré');

        // Le « flashBag » est ce qui contient les messages flash dans
        // la session
        // Il peut bien sûr contenir plusieurs messages :
        $this->get('session')->setFlashBag()->add('info', 'Oui oui, il
est bien enregistré !');

        // Puis on redirige vers la page de visualisation de cet
        // article
        return $this->redirect( $this->generateUrl('sdzblog_voir',
array('id' => 5)) );
    }
}
```

Vous pouvez voir que la méthode ajouterAction définit deux messages flash (appelés ici « info »). La lecture de ces messages se fait dans la vue de l'action voirAction, que j'ai modifiée comme ceci :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

<!DOCTYPE html>
<html>
    <head>
        <title>Bienvenue sur ma première page avec le Site du Zéro
    </title>
    </head>
    <body>
        <h1>Lecture d'un article</h1>

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

        <p>
            Ici nous pourrons lire l'article ayant comme id : {{ id }}<br>
        </p>
    </body>
</html>
```

```
/>
    Mais pour l'instant, nous ne savons pas encore le faire, cela
viendra !
</p>
</body>
</html>
```



La variable Twig `{ { app } }` est une variable globale, disponible partout dans vos vues. Elle contient quelques variables utiles, nous le verrons, dont le service « session » que nous venons d'utiliser via `{ { app.session } }`.

Essayez d'aller sur http://localhost/Symfony/web/app_dev.php/blog/ajouter, vous allez être redirigés et voir le message flash. Faites F5, et hop ! il a disparu.

Sachez également que le service « session » est aussi accessible depuis le service « request ». Ainsi, depuis un contrôleur vous pouvez faire :

Code : PHP

```
<?php
$session = $this->getRequest()->getSession();
```

Les autres... et les nôtres !

Il existe évidemment bien d'autres services : nous les renconterons au fur et à mesure dans ce cours.

Mais il existera surtout nos propres services ! En effet, la plupart des outils que nous allons créer (un formulaire, un gestionnaire d'utilisateurs personnalisé, etc.) devront être utilisés plusieurs fois. Quoi de mieux, dans ce cas, que de les définir en tant que services ? Nous verrons cela dans la partie 4, mais sachez qu'après une petite étape de mise en place (configuration, quelques conventions), les services sont vraiment très pratiques !

Application : le contrôleur de notre blog

Construction du contrôleur

Notre blog est un bundle plutôt simple. On va mettre toutes nos actions dans un seul contrôleur « Blog ». Plus tard, nous pourrons éventuellement créer un contrôleur « Tag » pour manipuler les tags.

Malheureusement, on ne connaît pas encore tous les services indispensables. À ce point du cours, on ne sait pas réaliser de formulaire, manipuler les articles dans la base de données, ni même créer de vrais templates.

Pour l'heure, notre contrôleur sera donc très simple. On va créer la base de toutes les actions que l'on a mises dans nos routes. Je vous remets sous les yeux nos routes, et on enchaîne sur le contrôleur :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_accueil:
    path:      /{page}
    defaults: { _controller: SdzBlogBundle:Blog:index, page: 1 }
    requirements:
        page: \d*

sdzblog_voir:
    path:      /article/{id}
    defaults: { _controller: SdzBlogBundle:Blog:voir }
    requirements:
        id: \d+
```

```

sdzblog_ajouter:
    path:      /ajouter
    defaults: { _controller: SdzBlogBundle:Blog:ajouter }

sdzblog_modifier:
    path:      /modifier/{id}
    defaults: { _controller: SdzBlogBundle:Blog:modifier }
    requirements:
        id: \d+

sdzblog_supprimer:
    path:      /supprimer/{id}
    defaults: { _controller: SdzBlogBundle:Blog:supprimer }
    requirements:
        id: \d+

```

Et le contrôleur « Blog » :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function indexAction($page)
    {
        // On ne sait pas combien de pages il y a
        // Mais on sait qu'une page doit être supérieure ou égale à 1
        if( $page < 1 )
        {
            // On déclenche une exception NotFoundHttpException
            // Cela va afficher la page d'erreur 404 (on pourra
            personnaliser cette page plus tard d'ailleurs)
            throw $this->createNotFoundException('Page inexistante (page =
'. $page .)');
        }

        // Ici, on récupérera la liste des articles, puis on la passera
        au template

        // Mais pour l'instant, on ne fait qu'appeler le template
        return $this->render('SdzBlogBundle:Blog:index.html.twig');
    }

    public function voirAction($id)
    {
        // Ici, on récupérera l'article correspondant à l'id $id

        return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
            'id' => $id
        ));
    }

    public function ajouterAction()
    {
        // La gestion d'un formulaire est particulière, mais l'idée est
        la suivante :

        if( $this->get('request')->getMethod() == 'POST' )

```

```
{  
    // Ici, on s'occupera de la création et de la gestion du  
    formulaire  
  
    $this->get('session')->getFlashBag()->add('notice', 'Article  
bien enregistré');  
  
    // Puis on redirige vers la page de visualisation de cet  
article  
    return $this->redirect( $this->generateUrl('sdzblog_voir',  
array('id' => 5)) );  
}  
  
// Si on n'est pas en POST, alors on affiche le formulaire  
return $this->render('SdzBlogBundle:Blog:ajouter.html.twig');  
}  
  
public function modifierAction($id)  
{  
    // Ici, on récupérera l'article correspondant à $id  
  
    // Ici, on s'occupera de la création et de la gestion du  
formulaire  
  
    return $this->render('SdzBlogBundle:Blog:modifier.html.twig');  
}  
  
public function supprimerAction($id)  
{  
    // Ici, on récupérera l'article correspondant à $id  
  
    // Ici, on gérera la suppression de l'article en question  
  
    return $this->render('SdzBlogBundle:Blog:supprimer.html.twig');  
}
```

À retenir

L'erreur 404

Je vous ai donné un exemple qui vous montre comment déclencher une erreur 404. C'est quelque chose que l'on fera souvent, par exemple dès qu'un article n'existera pas, qu'un argument ne sera pas bon (page = 0), etc. Lorsque l'on déclenche cette exception, le noyau l'attrape et génère une belle page d'erreur 404. Vous pouvez aller voir l'annexe « [Comment personnaliser ses pages d'erreur](#) ».

La définition des méthodes

Nos méthodes vont être appelées par le noyau : elles doivent donc respecter le nom et les arguments que nous avons définis dans nos routes et se trouver dans le *scope* « public ». Vous pouvez bien entendu rajouter d'autres méthodes, par exemple pour exécuter une fonction que vous réutiliserez dans deux actions différentes. Dans ce cas, vous ne devez pas les suffixer de « Action » (afin de ne pas confondre).

Testons-le

Naturellement, seules les actions `index` et `voir` vont fonctionner, car nous n'avons pas créé les templates associés (ce sera fait dans le prochain chapitre). Cependant, nous pouvons voir le type d'erreur que Symfony2 nous génère. Allez sur la page de suppression d'un article, à l'adresse http://localhost/Symfony/web/app_dev.php/blog/supprimer/5. Vous pouvez voir que l'erreur est très explicite et nous permet de voir directement ce qui ne va pas. On a même les logs en dessous de l'erreur : on peut voir tout ce qui a fonctionné avant que l'erreur ne se déclenche. Notez par exemple le log n°4 :

Code : Console

```
Matched route "sdzblog_supprimer" (parameters: "_controller": "Sdz\BlogBundle\Contr
```



On voit que c'est bien la bonne route qui est utilisée, super ! On voit aussi que le paramètre `id` est bien défini à 5 : re-super !

On peut également tester notre erreur 404 générée manuellement lorsque ce paramètre `page` est à 0. Allez sur http://localhost/Symfony/web/app_dev.php/blog/0, et admirez notre erreur. Regardez entre autres la *toolbar* (voir figure suivante).



Très pratique pour vérifier que tout est comme on l'attend !

Pour conclure

Créer un contrôleur à ce stade du cours n'est pas évident, car vous ne connaissez et ne maîtrisez pas encore tous les services nécessaires. Seulement, vous avez pu comprendre son rôle et voir un exemple concret.

Rassurez-vous, dans la partie 4 du tutoriel, on apprendra tout le nécessaire pour construire l'intérieur de nos contrôleurs. 😊
En attendant, rendez-vous au prochain chapitre pour en apprendre plus sur les templates.

Pour plus d'informations concernant les contrôleurs, n'hésitez pas à lire la [documentation officielle](#).

En résumé

- Le rôle du contrôleur est de retourner un objet `Response` : ceci est obligatoire !
- Le contrôleur construit la réponse en fonction des données qu'il a en entrée : paramètre de route et objet `Request`.
- Le contrôleur se sert de tout ce dont il a besoin pour construire la réponse : la base de données, les vues, les différents services, etc.

Le moteur de templates Twig

Les templates, ou vues, sont très intéressants. Nous l'avons déjà vu, leur objectif est de séparer le code PHP du code HTML. Ainsi, lorsque vous faites du PHP, vous n'avez pas 100 balises HTML qui gênent la lecture de votre code PHP. De même, lorsque votre designer fait du HTML, il n'a pas à subir votre code barbare PHP auquel il ne comprend rien.

Intéressé ? Lisez la suite. 

Les templates Twig

Intérêt

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

- La syntaxe est plus concise et plus claire. Rappelez-vous, pour afficher une variable, `{ { mavar } }` suffit, alors qu'en PHP il faudrait faire `<?php echo $mavar; ?>`.
- Il y a quelques fonctionnalités en plus, comme l'héritage de templates (nous le verrons). Cela serait bien entendu possible en PHP, mais il faudrait coder soi-même le système et cela ne serait pas aussi esthétique.
- Il sécurise vos variables automatiquement : plus besoin de se soucier de `htmlentities()`, `addslashes()` ou que sais-je encore.



Pour ceux qui se posent la question de la rapidité : aucune inquiétude ! Oui, il faut transformer le langage Twig en PHP avant de l'exécuter pour, finalement, afficher notre contenu. Mais Twig ne le fait que la première fois et met en cache du code PHP simple afin que, dès la deuxième exécution de votre page, ce soit en fait aussi rapide que du PHP simple.

Des pages web, mais aussi des e-mails et autres

En effet, pourquoi se limiter à nos pages HTML ? Les templates peuvent (et doivent) être utilisés partout. Quand on enverra des e-mails, leurs contenus seront placés dans un template. Il existe bien sûr un moyen de récupérer le contenu d'un template sans l'afficher immédiatement. Ainsi, en récupérant le contenu du template dans une variable quelconque, on pourra le passer à la fonction mail de notre choix.

Mais il en va de même pour un flux RSS par exemple ! Si l'on sait afficher une liste des news de notre site en HTML grâce au template `liste_news.html.twig`, alors on saura afficher un fichier RSS en gardant le même contrôleur, mais en utilisant le template `liste_news.rss.twig` à la place.

En pratique

On a déjà créé un template, mais un rappel ne fait pas de mal. Depuis le contrôleur, voici la syntaxe pour retourner une réponse HTTP toute faite, dont le contenu est celui d'un certain template :

Code : PHP

```
<?php
// Depuis un contrôleur

return $this->render('SdzBlogBundle:Blog:index.html.twig', array(
    'var1' => $var1,
    'var2' => $var2
));
```

Et voici comment, au milieu d'un contrôleur, récupérer le contenu d'un template en texte :

Code : PHP

```
<?php
// Depuis un contrôleur

$contenu = $this->renderView('SdzBlogBundle:Blog:email.txt.twig',
array(
    'var1' => $var1,
    'var2' => $var2
));

// Puis on envoie l'e-mail, par exemple :
mail('moi@siteduzero.com', 'Inscription OK', $contenu);
```

Et le template SdzBlogBundle:Blog:email.txt.twig contiendrait par exemple :

Code : Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/email.txt.twig #-}

Bonjour {{ pseudo }},

Toute l'équipe du site se joint à moi pour vous souhaiter la
bienvenue sur notre site !

Revenez nous voir souvent !
```

À savoir

Première chose à savoir sur Twig : *vous pouvez afficher des variables et pouvez exécuter des expressions*. Ce n'est pas la même chose :

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

L'objectif de la suite de ce chapitre est donc :

- D'abord, vous donner les outils pour afficher des variables : variables simples, tableaux, objets, appliquer des filtres, etc. ;
- Ensuite, vous donner les outils pour construire un vrai code dynamique : faire des boucles, des conditions, etc. ;
- Enfin, vous donner les outils pour organiser vos templates grâce à l'héritage et à l'inclusion de templates. Ainsi vous aurez un template maître qui contiendra votre design (avec les balises <html>, <head>, etc.) et vos autres templates ne contiendront que le contenu de la page (liste des news, etc.).

Afficher des variables

Syntaxe de base pour afficher des variables

Afficher une variable se fait avec les doubles accolades « {{ ... }} ». Voici quelques exemples.

Description	Exemple Twig	Équivalent PHP
Afficher une variable	Pseudo : {{ pseudo }}	Pseudo : <?php echo \$pseudo; ?>
Afficher l'index d'un tableau	Identifiant : {{ user['id'] }}	Identifiant : <?php echo \$user['id']; ?>
Afficher l'attribut d'un objet, dont le getter respecte la convention \$objet->getAttribut()	Identifiant : {{ user.id }}	Identifiant : <?php echo \$user->getId(); ?>

Afficher une variable en lui appliquant un filtre. Ici, « upper » met tout en majuscules :	Pseudo en majuscules : {{ pseudo upper }}	Pseudo en lettres majuscules : <?php echo strtoupper(\$pseudo); ?>
Afficher une variable en combinant les filtres. « striptags » supprime les balises HTML. « title » met la première lettre de chaque mot en majuscule. Notez l'ordre d'application des filtres, ici striptags est appliqué, puis title.	Message : {{ news.texte striptags title }}	Message : <?php echo ucwords(strip_tags(\$news->getTexte())); ?>
Utiliser un filtre avec des arguments. Attention, il faut que date soit un objet de type Datetime ici.	Date : {{ date date('d/m/Y') }}	Date : <?php echo \$date->format('d/m/Y'); ?>
Concaténer	Identité : {{ nom ~ " " ~ prenom }}	Identité : <?php echo \$nom.' '.\$prenom; ?>

Précisions sur la syntaxe {{ objet.attribut }}

Le fonctionnement de la syntaxe {{ objet.attribut }} est un peu plus complexe qu'elle n'en a l'air. Elle ne fait pas seulement objet->getAttribut. En réalité, voici ce qu'elle fait exactement :

- Elle vérifie si `objet` est un tableau, et si `attribut` est un index valide. Si c'est le cas, elle affiche `objet['attribut']`.
- Sinon, et si `objet` est un objet, elle vérifie si `attribut` est un attribut valide (public donc). Si c'est le cas, elle affiche `objet->attribut`.
- Sinon, et si `objet` est un objet, elle vérifie si `attribut()` est une méthode valide (publique donc). Si c'est le cas, elle affiche `objet->attribut()`.
- Sinon, et si `objet` est un objet, elle vérifie si `getAttribut()` est une méthode valide. Si c'est le cas, elle affiche `objet->getAttribut()`.
- Sinon, et si `objet` est un objet, elle vérifie si `isAttribut()` est une méthode valide. Si c'est le cas, elle affiche `objet->isAttribut()`.
- Sinon, elle n'affiche rien et retourne null.

Les filtres utiles

Il y a quelques filtres disponibles nativement avec Twig, en voici quelques-uns :

Filtre	Description	Exemple Twig
Upper	Met toutes les lettres en majuscules.	{{ var upper }}
Striptags	Supprime toutes les balises XML.	{{ var striptags }}
Date	Formate la date selon le format donné en argument. La variable en entrée doit être une instance de Datetime.	{{ date date('d/m/Y') }} Date d'aujourd'hui : {{ "now" date('d/m/Y') }}
Format	Insère des variables dans un texte, équivalent à <code>printf</code> .	{{ "Il y a %s pommes et %s poires" format(153, nb_poires) }}
Length	Retourne le nombre d'éléments du tableau, ou le nombre de caractères d'une chaîne.	Longueur de la variable : {{ texte length }} Nombre d'éléments du tableau : {{ tableau length }}



La documentation de tous les filtres disponibles est dans la documentation officielle de Twig :



<http://twig.sensiolabs.org/doc/filters/index.html>

Nous pourrons également créer nos propres filtres ! On le verra plus loin dans ce cours.

Twig et la sécurité

Dans tous les exemples précédents, vos variables ont déjà été protégées par Twig ! Twig applique par défaut un filtre sur toutes les variables que vous affichez, afin de les protéger de balises HTML malencontreuses. Ainsi, si le pseudo d'un de vos membres contient un « < » par exemple, lorsque vous faites {{ pseudo }} celui-ci est échappé, et le texte généré est en réalité « mon<pseudo » au lieu de « mon<pseudo », ce qui poserait problème dans votre structure HTML. Très pratique ! Et donc à savoir : inutile de protéger vos variables en amont, Twig s'occupe de tout en fin de chaîne !

Et dans le cas où vous voulez afficher volontairement une variable qui contient du HTML (JavaScript, etc.), et que vous ne voulez pas que Twig l'échappe, il vous faut utiliser le filtre `raw` comme suit : {{ ma_variable_html|raw }}. Avec ce filtre, Twig désactivera localement la protection HTML, et affichera la variable en brut, quel que soit ce qu'elle contient.

Les variables globales

Symfony2 enregistre quelques variables globales dans Twig pour nous faciliter la vie. Voici la liste des variables globales disponibles dans tous vos templates :

Variable	Description
{{ app.request }}	Le service « request » qu'on a vu au chapitre précédent sur les contrôleurs.
{{ app.session }}	Le service « session » qu'on a vu également au chapitre précédent.
{{ app.environment }}	L'environnement courant : « dev », « prod », et ceux que vous avez définis.
{{ app.debug }}	True si le mode debug est activé, False sinon.
{{ app.security }}	Le service « security », que nous verrons plus loin dans ce cours.
{{ app.user }}	L'utilisateur courant, que nous verrons également plus loin dans ce cours.

Bien entendu, on peut enregistrer nos propres variables globales, pour qu'elles soient accessibles depuis toutes nos vues, au lieu de les injecter à chaque fois. Pour cela, il faut éditer le fichier de configuration de l'application, comme suit :

Code : YAML

```
# app/config/config.yml

# ...

twig:
  # ...
  globals:
    webmaster: moi-même
```

Ainsi, la variable {{ webmaster }} sera injectée dans toutes vos vues, et donc utilisable comme ceci :

Code : HTML & Django

```
<footer>Responsable du site : {{ webmaster }}.</footer>
```

Je profite de cet exemple pour vous faire passer un petit message. Pour ce genre de valeurs paramétrables, la bonne pratique est de les définir non pas directement dans le fichier de configuration `config.yml`, mais dans le fichier des paramètres, à savoir `parameters.yml`. Attention, je parle bien de la valeur du paramètre, non de la configuration. Voyez par vous-mêmes.

Valeur du paramètre :

Code : YAML

```
# app/config/parameters.yml

parameters:
    #
    app_webmaster: moi-même
```

Configuration (ici, injection dans toutes les vues) qui utilise le paramètre :

Code : YAML

```
# app/config/config.yml

twig:
    globals:
        webmaster: %app_webmaster%
```

On a ainsi séparé la valeur du paramètre, stockée dans un fichier simple, et l'utilisation de ce paramètre, perdue dans le fichier de configuration.

Structures de contrôle et expressions

Les structures de contrôle

Nous avons vu comment *afficher* quelque chose, maintenant nous allons *faire* des choses, avec la syntaxe `{% ... %}`.

Condition : `{% if %}`

Exemple Twig :

Code : Django

```
{% if membre.age < 12 %}
Il faut avoir 12 ans pour ce film.
{% elseif membre.age < 18 %}
OK bon film.
{% else %}
Un peu vieux pour voir ce film non ?
{% endif %}
```

Équivalent PHP :

Code : PHP

```
<?php if($membre->getAge() < 12) { ?>
    Il faut avoir 12 ans pour ce film.
<?php } elseif($membre->getAge() < 18) { ?>
    OK bon film.
<?php } else { ?>
    Un peu vieux pour voir ce film non ?
<?php } ?>
```

Boucle : { % for %}

Exemple Twig :

Code : HTML & Django

```
<ul>
  { % for membre in liste_membres %}
    <li>{{ membre.pseudo }}</li>
  { % else %}
    <li>Pas d'utilisateur trouvé.</li>
  { % endfor %}
</ul>
```

Et pour avoir accès aux clés du tableau :

Code : HTML & Django

```
<select>
  { % for valeur, option in liste_options %}
    <option value="{{ valeur }}">{{ option }}</option>
  { % endfor %}
</select>
```

Équivalent PHP :

Code : PHP

```
<ul>
<?php if(count($liste_membres) > 0) {
  foreach($liste_membres as $membre) {
    echo '<li>' . $membre->getPseudo() . '</li>';
  }
} else { ?>
  <li>Pas d'utilisateur trouvé.</li>
<?php } ?>
</ul>
```

Avec les clés :

Code : PHP

```
<?php
foreach($liste_options as $valeur => $option) {
  // ...
}
```

Définition : { % set %}

Exemple Twig :

Code : Django

```
{ % set foo = 'bar' %}
```

Équivalent PHP :

Code : PHP

```
<?php $foo = 'bar'; ?>
```

Une petite information sur la structure `{% for %}`, celle-ci définit une variable `{ { loop } }` au sein de la boucle, qui contient les attributs suivants :

Variable	Description
<code>{ { loop.index } }</code>	Le numéro de l'itération courante (en commençant par 1).
<code>{ { loop.index0 } }</code>	Le numéro de l'itération courante (en commençant par 0).
<code>{ { loop.revindex } }</code>	Le nombre d'itérations restantes avant la fin de la boucle (en finissant par 1).
<code>{ { loop.revindex0 } }</code>	Le nombre d'itérations restantes avant la fin de la boucle (en finissant par 0).
<code>{ { loop.first } }</code>	<code>true</code> si c'est la première itération, <code>false</code> sinon.
<code>{ { loop.last } }</code>	<code>true</code> si c'est la dernière itération, <code>false</code> sinon.
<code>{ { loop.length } }</code>	Le nombre total d'itérations dans la boucle.

Les tests utiles

Defined

Pour vérifier si une variable existe.

Exemple Twig :

Code : Django

```
{% if var is defined %} ... {% endif %}
```

Équivalent PHP :

Code : PHP

```
<?php if(isset($var)) { } ?>
```

Even / Odd

Pour tester si un nombre est pair / impair.

Exemple Twig :

Code : HTML & Django

```
{% for valeur in liste %}
  <span class="{% if loop.index is even %}pair{% else %}
  impair{% endif %}">
    {{ valeur }}
  </span>
```

```
{% endfor %}
```

Équivalent PHP :

Code : PHP

```
<?php
$i = 0;
foreach($liste as $valeur) {
    echo '<span class=""';
    echo $i % 2 ? 'impair' : 'pair';
    echo '>' . $valeur . '</span>';
    $i++;
}
```



La documentation de tous les tests disponibles est dans la documentation officielle de Twig.

Hériter et inclure des templates

L'héritage de template

Je vous ai fait un *teaser* précédemment : l'héritage de templates va nous permettre de résoudre la problématique : « J'ai un seul design et n'ai pas l'envie de le répéter sur chacun de mes templates ». C'est un peu comme ce que vous devez faire aujourd'hui avec les `include()`, mais en mieux !

Le principe

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

L'avantage est que les templates fils peuvent modifier plusieurs blocs du template père. Avec la technique des `include()`, un template inclus ne pourra pas modifier le template père dans un autre endroit que là où il est inclus !

Les blocs classiques sont le centre de la page et le titre. Mais en fait, c'est à vous de les définir ; vous en ajouterez donc autant que vous voudrez.

La pratique

Voici à quoi peut ressembler un template père (appelé plus communément *layout*). Mettons-le dans `src/Sdz/BlogBundle/Resources/views/layout.html.twig` :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/layout.html.twig #-}

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

        {% block body %}
        {% endblock %}

    </body>
```

```
</html>
```

Voici un de nos templates fils. Mettons-le dans
src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #-}

{%- extends "SdzBlogBundle::layout.html.twig" %}

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

{% block body %}
    OK, même s'il est pour l'instant un peu vide, mon blog sera trop
    bien !
{% endblock %}
```

Qu'est-ce que l'on vient de faire ?

Pour bien comprendre tous les concepts utilisés dans cet exemple très simple, détaillons un peu.

Le nom du template père

On a placé ce template dans views/layout.html.twig et non dans views/qqch/layout.html.twig. C'est tout à fait possible ! En fait, il est inutile de mettre dans un sous-répertoire les templates qui ne concernent pas un contrôleur particulier et qui peuvent être réutilisés par plusieurs contrôleurs. Attention à la notation pour accéder à ce template : du coup, ce n'est plus SdzBlogBundle:MonController:layout.html.twig, mais SdzBlogBundle::layout.html.twig. C'est assez intuitif, en fait : on enlève juste la partie qui correspond au répertoire MonController. C'est ce que l'on a fait à la première ligne du template fils.

La balise {%- block %} côté père

Pour définir un « trou » (dit *bloc*) dans le template père, nous avons utilisé la balise {%- block %}. Un bloc doit avoir un nom afin que le template fils puisse modifier tel ou tel bloc de façon nominative. La base, c'est juste de faire {%- block nom_du_bloc %}{%- endblock %} et c'est ce que nous avons fait pour le *body*. Mais vous pouvez insérer un texte par défaut dans les blocs, comme on l'a fait pour le *titre*. C'est utile pour deux cas de figure :

- Lorsque le template fils ne redéfinit pas ce bloc. Plutôt que de n'avoir rien d'écrit, vous aurez cette valeur par défaut.
- Lorsque les templates fils veulent réutiliser une valeur commune. Par exemple, si vous souhaitez que le titre de toutes les pages de votre site commence par « SdzBlog », alors depuis les templates fils, vous pouvez utiliser {{ parent() }} qui permet d'utiliser le contenu par défaut du bloc côté père. Regardez, nous l'avons fait pour le *titre* dans le template fils.

La balise {%- block %} côté fils

Elle se définit exactement comme dans le template père, sauf que cette fois-ci on y met notre contenu. Mais étant donné que les blocs se définissent et se remplissent de la même façon, vous avez pu deviner qu'on peut hériter en cascade ! En effet, si l'on crée un troisième template petit-fils qui hérite de fils, on pourra faire beaucoup de choses.

Le modèle « triple héritage »

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

- Layout général : c'est le design de votre site, indépendamment de vos bundles. Il contient le header, le footer, etc. La structure de votre site donc (c'est notre template père).
- Layout du bundle : il hérite du layout général et contient les parties communes à toutes les pages d'un même bundle. Par exemple, pour notre blog, on pourrait afficher un menu particulier, rajouter « Blog » dans le titre, etc.
- Template de page : il hérite du layout du bundle et contient le contenu central de votre page.

Nous verrons un exemple de ce triple héritage juste après dans l'exemple du blog.

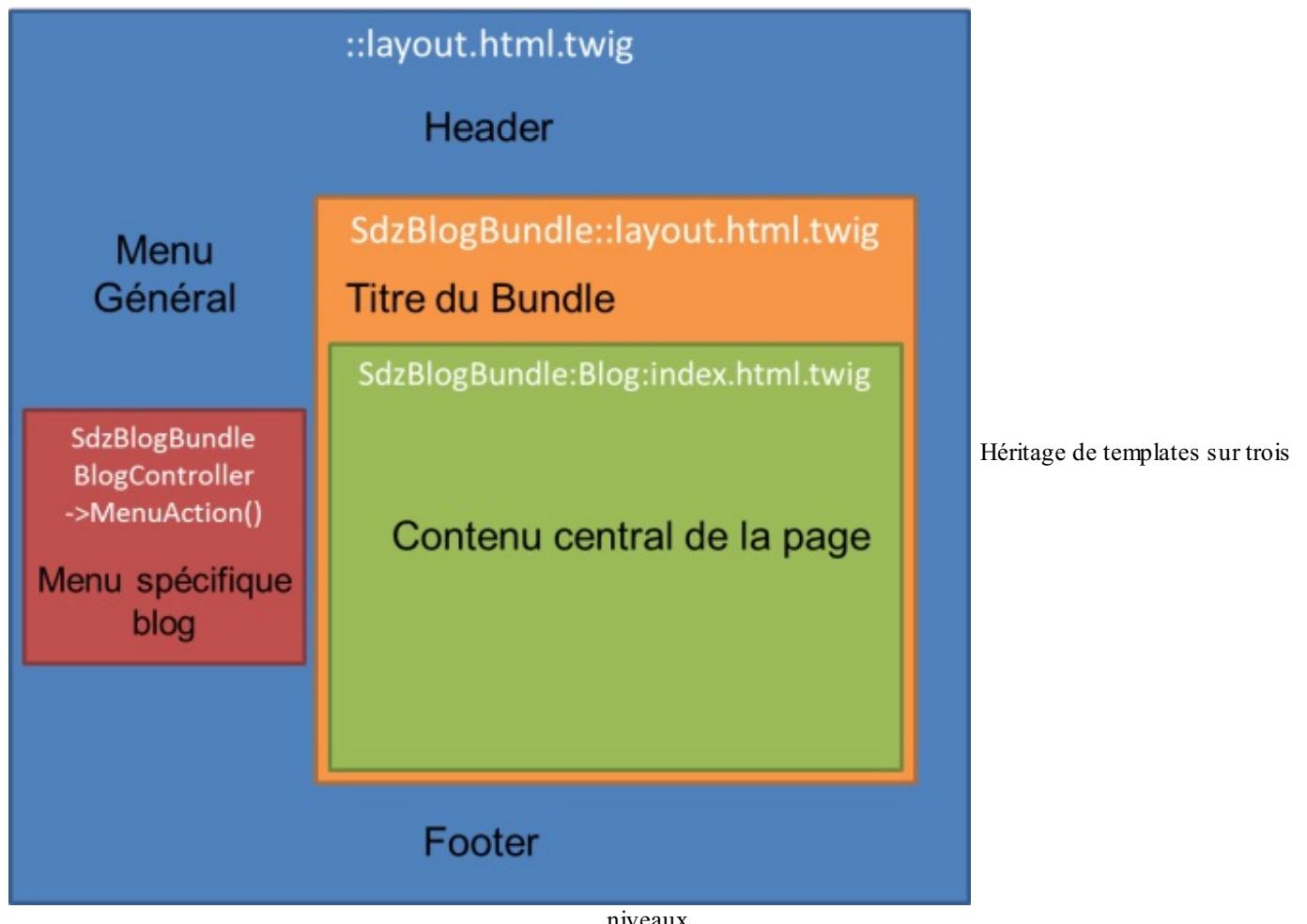


Question : puisque le layout général ne dépend pas d'un bundle en particulier, où le mettre ?

Dans votre répertoire /app ! En effet, dans ce répertoire, vous pouvez toujours avoir des fichiers qui écrasent ceux des bundles ou bien des fichiers communs aux bundles. Le layout général de votre site fait partie de ces ressources communes. Son répertoire exact doit être app/Resources/views/layout.html.twig.

Et pour l'appeler depuis vos templates, la syntaxe est la suivante : « ::layout.html.twig ». Encore une fois, c'est très intuitif : après avoir enlevé le nom du contrôleur tout à l'heure, on enlève juste cette fois-ci le nom du bundle.

Afin de bien vous représenter l'architecture adoptée, je vous propose un petit schéma à la figure suivante. Il vaut ce qu'il vaut, mais vous permet de bien comprendre ce qu'on fait.



Je vous parle du bloc rouge un peu après, c'est une inclusion non pas de template, mais d'action de contrôleur ! Il ne fait pas partie du modèle triple héritage à proprement parler.

L'inclusion de templates

La théorie : quand faire de l'inclusion ?

Hériter, c'est bien, mais inclure, cela n'est pas mal non plus. Prenons un exemple pour bien faire la différence.

Le formulaire pour ajouter un article est le même que celui pour... modifier un article. On ne va pas faire du copier-coller de code, cela serait assez moche, et puis nous sommes fainéants. C'est ici que l'inclusion de templates intervient. On a nos deux templates `SdzBlogBundle:Blog:ajouter.html.twig` et `SdzBlogBundle:Blog:modifier.html.twig` qui héritent chacun de `SdzBlogBundle::layout.html.twig`. L'affichage exact de ces deux templates diffère un peu, mais chacun d'eux inclut `SdzBlogBundle:Blog:formulaire.html.twig` à l'endroit exact pour afficher le formulaire.

On voit bien qu'on ne peut pas faire d'héritage sur le template `formulaire.html.twig`, car il faudrait le faire hériter une fois de `ajouter.html.twig`, une fois de `modifier.html.twig`, etc. Comment savoir ? Et si un jour nous souhaitons ne le faire hériter de rien du tout pour afficher le formulaire tout seul dans une *popup* par exemple ? Bref, c'est bien une inclusion qu'il nous faut ici.

La pratique : comment le faire ?

Comme toujours avec Twig, cela se fait très facilement. Il faut utiliser la balise `{% include %}`, comme ceci :

Code : HTML & Django

```
{% include "SdzBlogBundle:Blog:formulaire.html.twig" %}
```

Ce code inclura le contenu du template à l'endroit de la balise. Une sorte de copier-coller automatique, en fait ! Voici un exemple avec la vue `ajouter.html.twig` :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/ajouter.html.twig #-}

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

{% block body %}

<h2>Ajouter un article</h2>

{% include "SdzBlogBundle:Blog:formulaire.html.twig" %}

<p>
    Attention : cet article sera ajouté directement
    sur la page d'accueil après validation du formulaire.
</p>

{% endblock %}
```

Et voici le code du template inclus :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/formulaire.html.twig #

# Cette vue n'hérite de personne, elle sera incluse par d'autres
# vues qui, elles, hériteront probablement du layout. #}
# Je dis « probablement » car, ici pour cette vue, on n'en sait
rien et c'est une info qui ne nous concerne pas. #}

<h3>Formulaire d'article</h3>
```

```
{# Ici on laisse vide la vue pour l'instant, on la comblera plus tard lorsqu'on saura afficher un formulaire. #}
<div class="well">
    Ici se trouvera le formulaire.
</div>
```



À l'intérieur du template inclus, vous retrouvez toutes les variables qui sont disponibles dans le template qui fait l'inclusion : exactement comme si vous copiez-collez le contenu.

L'inclusion de contrôleurs

La théorie : quand inclure des contrôleurs ?

Voici un dernier point à savoir absolument avec Twig, un des points les plus puissants dans son utilisation avec Symfony2. On vient de voir comment inclure des templates : ceux-ci profitent des variables du template qui fait l'inclusion, très bien.

Seulement dans bien des cas, depuis le template qui fait l'inclusion, vous voudrez inclure un autre template, mais vous n'avez pas les variables nécessaires pour lui. Restons sur l'exemple de notre blog, dans le schéma précédent je vous ai mis un bloc rouge : considérons que dans cette partie du menu, accessible sur toutes les pages même hors du blog, on veut afficher les 3 derniers articles du blog.

C'est donc depuis le layout général qu'on va inclure non pas un template du bundle Blog — nous n'aurions pas les variables à lui donner —, mais un contrôleur du bundle Blog. Le contrôleur va créer les variables dont il a besoin, et les donner à son template, pour ensuite être inclus là où on le veut !

La pratique : comment le faire ?

Au risque de me répéter : cela se fait très simplement !

Du côté du template qui fait l'inclusion, à la place de la balise `{% include %}`, il faut utiliser la fonction `{} render()`, comme ceci :

Code : HTML & Django

```
{ { render(controller("SdzBlogBundle:Blog:menu")) } }
```

Ici, `SdzBlogBundle:Blog:menu` n'est pas un template mais une action de contrôleur, c'est la syntaxe qu'on utilise dans les routes, vous l'aurez reconnue.



Si vous utilisez une version de Symfony antérieure à la 2.2, alors il faut utiliser la syntaxe suivante : `{% render "SdzBlogBundle:Blog:menu" %}` à la place du code précédent. Cela ne change strictement rien au comportement, c'est juste la syntaxe qui change. 😊

Voici par exemple ce qu'on mettrait dans le layout :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{ % block title %}SdzBlog{ % endblock %}</title>
```

```

</head>
<body>

    <div id="menu">
        {{ render(controller("SdzBlogBundle:Blog:menu")) }}
    </div>

    {%
        block body %
        %endblock %
    %}

    </body>
</html>

```

Et du côté du contrôleur, c'est une méthode très classique (regardez la ligne 16) :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...

public function menuAction()
{
    // On fixe en dur une liste ici, bien entendu par la suite on
    // la récupérera depuis la BDD !
    $liste = array(
        array('id' => 2, 'titre' => 'Mon dernier weekend !'),
        array('id' => 5, 'titre' => 'Sortie de Symfony2.1'),
        array('id' => 9, 'titre' => 'Petit test')
    );

    return $this->render('SdzBlogBundle:Blog:menu.html.twig', array(
        'liste_articles' => $liste // C'est ici tout l'intérêt : le
        // contrôleur passe les variables nécessaires au template !
    ));
}

```

Et enfin, un exemple de ce que pourrait être le template menu.html.twig :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/menu.html.twig #-}



<ul>
    <ul class="nav nav-pills nav-stacked">
        {% for article in liste_articles %}
            <li><a href="{{ path('sdzblog_voir', {'id': article.id}) }}>{{ article.titre }}</a></li>
        {% endfor %}
    </ul>
</ul>

```

Application : les templates de notre blog

Revenons à notre blog. Faites en sorte d'avoir sous la main le contrôleur que l'on a réalisé au chapitre précédent. Le but ici est de créer tous les templates que l'on a utilisés depuis le contrôleur, ou du moins leur squelette. Étant donné que l'on n'a pas encore la vraie liste des articles, on va faire avec des variables vides : cela va se remplir par la suite, mais le fait d'employer des variables vides va nous permettre dès maintenant de construire le template.

Pour encadrer tout ça, nous allons utiliser le modèle d'héritage sur trois niveaux : layout général, layout du bundle et template.

Layout général

La théorie

Comme évoqué précédemment, le layout est la structure HTML de notre site avec des blocs aux endroits stratégiques pour permettre aux templates qui hériteront de ce dernier de personnaliser la page. On va ici créer une structure simple ; je vous laisse la personnaliser si besoin est. Pour les blocs, pareil pour l'instant, on fait simple : un bloc pour le body et un bloc pour le titre.

Je vais également en profiter pour introduire l'utilisation de ressources CSS/JS/etc. dans Symfony2. Cela se fait très bien avec la fonction `{ { asset() } }` de Twig, qui va chercher vos ressources dans le répertoire `/web`. Regardez simplement comment elle est utilisée dans l'exemple et vous saurez l'utiliser de façon basique.

 Pour le design du blog que l'on va construire, je vais utiliser le [Bootstrap de Twitter](#). C'est un framework CSS, l'équivalent pour le CSS de ce que Symfony2 est pour le PHP. Cela permet de faire des beaux designs, boutons ou liens très rapidement. Vous pourrez le voir dans les vues que je fais par la suite. Mais tout d'abord, vous devez le télécharger et l'extraire dans le répertoire `/web`. Vous devez avoir le fichier `bootstrap.css` disponible dans le répertoire `/web/css/bootstrap.css` par exemple. Il existe [un cours sur Bootstrap sur le Site du Zéro](#). Je vous invite à jeter un œil si ce framework vous intéresse, ou juste par curiosité.

La pratique

Commençons par faire le layout général de l'application, la vue située dans le répertoire `/app`. Voici le code exemple que je vous propose :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

        <title>{ % block title %}Sdz{ % endblock %}</title>

        { % block stylesheets %}
            <link rel="stylesheet" href="{{ asset('css/bootstrap.css') }}"
type="text/css" />
        { % endblock %
    </head>

    <body>
        <div class="container">
            <div id="header" class="hero-unit">
                <h1>Mon Projet Symfony2</h1>
                <p>Ce projet est propulsé par Symfony2, et construit grâce au
tutoriel du siteduzero.</p>
                <p><a class="btn btn-primary btn-large"
href="http://www.siteduzero.com/informatique/tutoriels/developpez-votre-
site-web-avec-le-framework-symfony2">
                    Lire le tutoriel >
                </a></p>
            </div>

            <div class="row">
                <div id="menu" class="span3">
                    <h3>Le blog</h3>
                    <ul class="nav nav-pills nav-stacked">
                        <li><a href="{{ path('sdzblog_accueil') }}">Accueil du
blog</a></li>
```

```

<li><a href="{{ path('sdzblog_ajouter') }}">Ajouter un
article</a></li>
</ul>

    {{ render(controller("SdzBlogBundle:Blog:menu", { 'nombre': 3 })) }
}

</div>
<div id="content" class="span9">
    {%
        block body %
    %}
    {%
        endblock %
}
</div>
</div>

<hr>

<footer>
    <p>The sky's the limit © 2012 and beyond.</p>
</footer>
</div>

{%
    block javascripts %
    {# Ajoutez ces lignes JavaScript si vous comptez vous servir des
fonctionnalités du bootstrap Twitter #}
    <script
src="//ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script type="text/javascript" src="{{ asset('js/bootstrap.js') }}"
} "></script>
    {%
        endblock %
}

</body>
</html>

```

Voici les lignes qui contiennent un peu de Twig :

- Ligne 8 : création du bloc « title » avec « Sdz » comme contenu par défaut ;
- Lignes 11 : appel du CSS situé dans /web/css/bootstrap.css ;
- Lignes 29 et 30 : utilisation de la fonction {{ path }} pour faire des liens vers d'autres routes ;
- Ligne 33 : inclusion de la méthode menu du contrôleur Blog du bundle SdzBlogBundle, avec l'argument nombre défini à 3 ;
- Lignes 36 et 37 : création du bloc « body » sans contenu par défaut.

Et voilà, nous avons notre layout général ! Pour pouvoir tester nos pages, il faut maintenant s'attaquer au layout du bundle.

Layout du bundle

La théorie

Comme on l'a dit, ce template va hériter du layout général, ajouter la petite touche personnelle au bundle Blog, puis se faire hériter à son tour par les templates finaux. En fait, il ne contient pas grand-chose. Laissez courir votre imagination, mais, moi, je ne vais rajouter qu'une balise **<h1>**, vous voyez ainsi le mécanisme et pouvez personnaliser à votre sauce.

La seule chose à laquelle il faut faire attention, c'est au niveau du nom des blocs que ce template crée pour ceux qui vont l'hériter. Une bonne pratique consiste à préfixer le nom des blocs par le nom du bundle courant. Regardez le code et vous comprendrez.

La pratique

Voici ce que j'ai mis pour le layout du bundle :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/layout.html.twig #}
```

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

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

{% block body %}

    {# On définit un sous-titre commun à toutes les pages du bundle,
    par exemple #}
    <h1>Blog</h1>

    <hr>

    {# On définit un nouveau bloc, que les vues du bundle pourront
    remplir #}
    {% block sdzblog_body %}
    {% endblock %}

    {% endblock %}
```

On a ajouté un **<h1>** dans le bloc **body**, puis créé un nouveau bloc qui sera personnalisé par les templates finaux. On a préfixé le nom du nouveau bloc pour le **body** afin d'avoir un nom unique pour notre bundle.

Les templates finaux

Blog/index.html.twig

C'est le template de la page d'accueil. On va faire notre première boucle sur la variable `{{ articles }}`. Cette variable n'existe pas encore, on va modifier le contrôleur juste après.

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #-}

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

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

{% block sdzblog_body %}

    <h2>Liste des articles</h2>

    <ul>
        {% for article in articles %}
        <li>
            <a href="{{ path('sdzblog_voir', {'id': article.id}) }}>{{ article.titre }}</a>
            par {{ article.auteur }},
            le {{ article.date|date('d/m/Y') }}
        </li>
        {% else %}
        <li>Pas (encore !) d'articles</li>
        {% endfor %}
    </ul>

    {% endblock %}
```

Pas grand-chose à dire, on a juste utilisé les variables et expressions expliquées dans ce chapitre.

Afin que cette page fonctionne, il nous faut modifier l'action `indexAction()` du contrôleur pour passer une variable `{ "articles" } à cette vue. Pour l'instant, voici juste de quoi se débarrasser de l'erreur :`

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// Dans l'action indexAction() :
return $this->render('SdzBlogBundle:Blog:index.html.twig', array(
    'articles' => array()
));
```



Vous pouvez dès maintenant voir votre nouvelle peau :
http://localhost/Symfony/web/app_dev.php/blog !

Si vous n'aviez pas rajouté l'action `menu` du contrôleur tout à l'heure, voici comment le faire, et aussi comment l'adapter à l'argument qu'on lui a passé cette fois-ci :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function menuAction($nombre) // Ici, nouvel argument
$nombre, on l'a transmis via le render() depuis la vue
{
    // On fixe en dur une liste ici, bien entendu par la suite on
    // la récupérera depuis la BDD !
    // On pourra récupérer $nombre articles depuis la BDD,
    // avec $nombre un paramètre qu'on peut changer lorsqu'on
    // appelle cette action
    $liste = array(
        array('id' => 2, 'titre' => 'Mon dernier weekend !!'),
        array('id' => 5, 'titre' => 'Sortie de Symfony2.1'),
        array('id' => 9, 'titre' => 'Petit test')
    );

    return $this->render('SdzBlogBundle:Blog:menu.html.twig', array(
        'liste_articles' => $liste // C'est ici tout l'intérêt : le
        // contrôleur passe les variables nécessaires au template !
    ));
}
```

Avec sa vue associée :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/menu.html.twig #-}

<h3>Les derniers articles</h3>

<ul class="nav nav-pills nav-stacked">
    {% for article in liste_articles %}
        <li><a href="{{ path('sdzblog_voir', { 'id': article.id}) }}>{{ article.titre }}</a></li>
    {% endfor %}
</ul>
```

Vous voulez voir des articles au lieu du message pas très drôle comme quoi il n'y a pas encore d'article ? Voici un tableau d'articles à ajouter temporairement dans la méthode `indexAction()`, que vous pouvez passer en paramètre à la méthode `render()`. C'est un tableau pour l'exemple, par la suite il faudra bien sûr récupérer les articles depuis la base de données.

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...

public function indexAction()
{
    // ...

    // Les articles :
    $articles = array(
        array(
            'titre'    => 'Mon weekend a Phi Phi Island !',
            'id'       => 1,
            'auteur'   => 'winzou',
            'contenu'  => 'Ce weekend était trop bien. Blabla...',
            'date'     => new \Datetime(),
        array(
            'titre'    => 'Repetition du National Day de Singapour',
            'id'       => 2,
            'auteur'   => 'winzou',
            'contenu'  => 'Bientôt prêt pour le jour J. Blabla...',
            'date'     => new \Datetime(),
        array(
            'titre'    => 'Chiffre d\'affaire en hausse',
            'id'       => 3,
            'auteur'   => 'M@teo21',
            'contenu'  => '+500% sur 1 an, fabuleux. Blabla...',
            'date'     => new \Datetime()
        );
    );

    // Puis modifiez la ligne du render comme ceci, pour prendre en
    // compte nos articles :
    return $this->render('SdzBlogBundle:Blog:index.html.twig', array(
        'articles' => $articles
    ));
}
```

Rechargez la page, et profitez du résultat. 😊 Si vous avez bien ajouté le CSS de Twitter, le résultat devrait ressembler à la figure suivante.

Mon Projet Symfony2

Ce projet est propulsé par Symfony2, et construit grâce au tutoriel du siteduzero.

[Lire le tutoriel »](#)

Le blog

[Accueil du blog](#)

[Ajouter un article](#)

Les derniers articles

[Mon dernier weekend !](#)

[Sortie de Symfony2.1](#)

[Petit test](#)

Blog

Liste des articles

- [Mon weekend a Phi Phi Island !](#) par winzou, le 18/07/2012
- [Repetition du National Day de Singapour](#) par winzou, le 18/07/2012
- [Chiffre d'affaire en hausse](#) par M@teo21, le 18/07/2012

The sky's the limit © 2012 and beyond.

Le rendu de notre blog



Attention, on vient de définir des articles en brut dans le contrôleur, mais c'est uniquement pour l'exemple d'utilisation de Twig ! Ce n'est bien sûr pas du tout une façon correcte de le faire, par la suite nous les récupérerons depuis la base de données.

[Blog/voir.html.twig](#)

Il ressemble beaucoup à `index.html.twig` sauf qu'on passe à la vue une variable `{{ article }}` contenant un seul article, et non plus une liste d'articles. Voici un code par exemple :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

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

{%
    block title %
        Lecture d'un article - {{ parent() }}
{%
    endblock %}

{%
    block sdzblog_body %
        

## {{ article.titre }}

Par {{ article.auteur }}, le {{ article.date|date('d/m/Y') }}

{{ article.contenu }}


        Retour à la liste
    {/
}
```

```
<a href="{{ path('sdzblog_modifier', {'id': article.id}) }}"  
class="btn">  
    <i class="icon-edit"></i>  
    Modifier l'article  
</a>  
<a href="{{ path('sdzblog_supprimer', {'id': article.id}) }}"  
class="btn">  
    <i class="icon-trash"></i>  
    Supprimer l'article  
</a>  
</p>  
  
{% endblock %}
```

Et l'adaptation du contrôleur bien évidemment :

Code : PHP

```
<?php  
// src/Sdz/BlogBundle/Controller/BlogController.php  
  
// ...  
  
public function voirAction($id)  
{  
    // ...  
  
    $article = array(  
        'id'      => 1,  
        'titre'   => 'Mon weekend a Phi Phi Island !',  
        'auteur'  => 'winzou',  
        'contenu' => 'Ce weekend était trop bien. Blabla...',  
        'date'    => new \Datetime()  
    );  
  
    // Puis modifiez la ligne du render comme ceci, pour prendre en  
    // compte l'article :  
    return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(  
        'article' => $article  
    ));  
}
```

La figure suivante représente le rendu de /blog/article/1.

Mon Projet Symfony2

Ce projet est propulsé par Symfony2, et construit grâce au tutoriel du siteduzero.

[Lire le tutoriel »](#)

Le blog

[Accueil du blog](#)

[Ajouter un article](#)

Les derniers articles

[Mon dernier weekend !](#)

[Sortie de Symfony2.1](#)

[Petit test](#)

Blog

Mon weekend a Phi Phi Island !

Par winzou, le 18/07/2012

Ce weekend était trop bien. Blabla...

[◀ Retour à la liste](#)

[Modifier l'article](#)

[Supprimer l'article](#)

The sky's the limit © 2012 and beyond.

Visualisation d'un article

[Blog/modifier.html.twig et ajouter.html.twig](#)

Ceux-ci contiennent une inclusion de template. En effet, rappelez-vous, j'avais pris l'exemple d'un formulaire utilisé pour l'ajout, mais également la modification. C'est notre cas ici, justement. Voici donc le fichier `modifier.html.twig`:

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/modifier.html.twig #-}

{%- extends "SdzBlogBundle::layout.html.twig" %}

{%- block title %}
    Modifier un article - {{ parent() }}
{%- endblock %}

{%- block sdzblog_body %}

    <h2>Modifier un article</h2>

    {%- include "SdzBlogBundle:Blog:formulaire.html.twig" %}

    <p>
        Vous éditez un article déjà existant,
        ne le changez pas trop pour éviter
        aux membres de ne pas comprendre
        ce qu'il se passe.
    </p>

    <p>
        <a href="{{ path('sdzblog_voir', {'id': article.id}) }}" class="btn">
            <i class="icon-chevron-left"></i>
            Retour à l'article
        </a>
    </p>

```

```
{% endblock %}
```

Le template ajouter.html.twig lui ressemble énormément, je vous laisse donc le faire.

Quant à formulaire.html.twig, on ne sait pas encore le faire, car il demande des notions de formulaire, mais faisons déjà sa structure pour le moment :

Code : HTML & Django

```
# src/Sdz/BlogBundle/Resources/views/Blog/formulaire.html.twig #}

{# Cette vue n'hérite de personne, elle sera incluse par d'autres
vues qui, elles, hériteront probablement du layout #}
{# Je dis « probablement » car, ici pour cette vue, on n'en sait
rien et c'est une info qui ne nous concerne pas #}

<h3>Formulaire d'article</h3>

{# Ici on laisse vide la vue pour l'instant, on la comblera plus
tard lorsqu'on saura afficher un formulaire #}
<div class="well">
    Ici se trouvera le formulaire.
</div>
```

Une chose importante ici : dans ce template, il n'y a aucune notion de bloc, d'héritage, etc. Ce template est un électron libre : vous pouvez l'inclure depuis n'importe quel autre template.

Et, bien sûr, il faut adapter le contrôleur pour passer la variable article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function modifierAction($id)
{
    // Ici, on récupérera l'article correspondant à $id

    // Ici, on s'occupera de la création et de la gestion du
formulaire

    $article = array(
        'id'      => 1,
        'titre'   => 'Mon weekend a Phi Phi Island !',
        'auteur'  => 'winzou',
        'contenu' => 'Ce weekend était trop bien. Blabla...',
        'date'    => new \Datetime()
    );

    // Puis modifiez la ligne du render comme ceci, pour prendre en
compte l'article :
    return $this->render('SdzBlogBundle:Blog:modifier.html.twig',
array(
    'article' => $article
));
}
```

Ainsi, /blog/modifier/1 nous donnera la figure suivante.

Mon Projet Symfony2

Ce projet est propulsé par Symfony2, et construit grâce au tutoriel du siteduzero.

[Lire le tutoriel »](#)

Le blog

[Accueil du blog](#)

[Ajouter un article](#)

Les derniers articles

[Mon dernier weekend !](#)

[Sortie de Symfony2.1](#)

[Petit test](#)

Blog

Modifier un article

Formulaire d'article

Ici se trouvera le formulaire.

Vous éditez un article déjà existant, ne le changez pas trop pour éviter aux membres de ne pas comprendre ce qu'il se passe.

[◀ Retour à l'article](#)

The sky's the limit © 2012 and beyond.

Modification d'un article

Pour conclure

Et voilà, nous avons généré presque tous nos templates. Bien sûr, ils sont encore un peu vides, car on ne sait pas utiliser les formulaires ni récupérer les articles depuis la base de données. Mais vous savez maintenant les réaliser et c'était une étape importante ! Je vais vous laisser créer les templates manquants ou d'autres afin que vous vous fassiez la main. Bon code !

Cela termine ce chapitre : vous savez afficher avec mise en forme le contenu de votre site. Vous avez maintenant presque toutes les billes en main pour réaliser un site internet. Bon, OK, c'est vrai, il vous manque encore des concepts clés tels que les formulaires, la base de données, etc. Mais vous maîtrisez pleinement la base du framework Symfony2, et apprendre ces prochains concepts sera bien plus facile !

Pour plus d'informations concernant Twig et ses possibilités, n'hésitez pas à lire la [documentation officielle](#).

En résumé

- Un moteur de templates tel que Twig permet de bien séparer le code PHP du code HTML, dans le cadre de l'architecture MVC ;
- La syntaxe `{ { var } }` affiche la variable var ;
- La syntaxe `{% if %}` fait quelque chose, ici une condition ;
- Twig offre un système d'héritage (via `{% extends %}`) et d'inclusion (via `{% include %}` et `{% render %}`) très intéressant pour bien organiser les templates ;
- Le modèle triple héritage est très utilisé pour des projets avec Symfony2.

Installer un bundle grâce à Composer

Je fais une parenthèse pour vous présenter un chapitre sur l'outil Composer.

Cet outil ne fait absolument pas partie de Symfony2, mais son usage est tellement omniprésent dans la communauté Symfony2 que je me dois de vous le présenter. Nous faisons donc une pause dans le développement de notre blog pour parler un peu de cet outil de gestion de dépendances, qui va nous servir à installer des bundles et autres bibliothèques très facilement.

Bonne lecture !

Composer, qu'est-ce que c'est ? Un gestionnaire de dépendances

Composer est un outil pour gérer les dépendances en PHP. Les dépendances, dans un projet, ce sont toutes les bibliothèques dont votre projet dépend pour fonctionner. Par exemple, votre projet utilise la bibliothèque SwiftMailer pour envoyer des e-mails, il « dépend » donc de SwiftMailer. Autrement dit, SwiftMailer est une dépendance dans votre projet.

Composer a donc pour objectif de vous aider à gérer toutes vos dépendances. En effet, il y a plusieurs problématiques lorsqu'on utilise des bibliothèques externes :

- Ces bibliothèques sont mises à jour. Il vous faut donc les mettre à jour une à une pour vous assurer de corriger les bogues de chacune d'entre elles.
- Ces bibliothèques peuvent elles-mêmes dépendre d'autres bibliothèques. En effet, si une de vos bibliothèques dépend d'autres bibliothèques, cela vous oblige à gérer l'ensemble de ces dépendances (installation, mises à jour, etc.).
- Ces bibliothèques ont chacune leur paramètres d'*autoload*, et vous devez gérer leur *autoload* pour chacune d'entre elles.

Composer va nous aider dans chacune de ces tâches.

Comment Composer sait où trouver les bibliothèques ?

Très bonne question. En effet, il est évident que ce système de gestion ne peut marcher que si on peut centraliser les informations de chaque bibliothèque. Il s'agit du site www.packagist.org.

Par exemple, voici la page pour la bibliothèque Symfony2 (eh oui, c'est une bibliothèque comme une autre !) : <https://packagist.org/packages/symfony/symfony>. Vous pouvez voir les informations comme le mainteneur principal, le site de la bibliothèque, etc. Mais ce qui nous importe le plus, ce sont les sources ainsi que les dépendances (dans Requires).

Composer va donc lire ces informations, et a alors toutes les cartes en main pour télécharger Symfony2 ainsi que ses dépendances.

Un outil innovant... dans l'écosystème PHP

Ce genre d'outil de gestion de dépendances n'est pas du tout nouveau dans le monde informatique. Vous connaissez peut-être déjà APT, le gestionnaire de paquets de la distribution Linux Debian. Il existe également des outils de ce genre pour le langage Ruby par exemple. Cependant, jusque très récemment, il n'existe aucun outil de ce genre pour PHP. La forte communauté qui s'est construite autour de Symfony2 a fait naître le besoin d'un tel outil, et l'a ensuite développé.

Concrètement, comment ça marche ?

Concrètement, voici comment s'utilise Composer :

- On définit dans un fichier la liste des bibliothèques dont le projet dépend, ainsi que leur version ;
- On exécute une commande pour installer ou mettre à jour ces bibliothèques (et leurs propres dépendances donc) ;
- On inclut alors le fichier d'*autoload* généré par Composer dans notre projet.

Mais avant de manipuler Composer, commençons par l'installer !

Installer Composer et Git

Installer Composer

Installer Composer est très facile, il suffit d'une seule commande... PHP ! Exécutez cette commande dans la console :

Code : Console

```
C:\wamp\www> php -r "eval('?
>'.file_get_contents('http://getcomposer.org/installer'));"
```

Cette commande va télécharger et exécuter le fichier PHP suivant : <http://getcomposer.org/installer>. Vous pouvez aller le voir, ce n'est pas Composer en lui-même mais son installateur. Il fait quelques vérifications (version de PHP, etc.), puis télécharge effectivement Composer dans le fichier `composer.phar`.

Composer en lui-même, c'est le fichier PHAR `composer.phar`, c'est lui que nous devrons exécuter par la suite. Vous pouvez déjà l'exécuter pour vérifier que tout est OK :

Code : Console

```
C:\wamp\www>php composer.phar --version
Composer version a5eaba8
```

N'hésitez pas à mettre à jour Composer lui-même de temps en temps. Il faut pour cela utiliser la commande `self-update` de Composer, comme suit :

Code : Console

```
C:\wamp\www>php composer.phar self-update
Updating to version ded485d.
Downloading: 100%
```

Et voilà, je viens de mettre à jour Composer très simplement !

Cependant, l'installation n'est pas finie. En effet, pour récupérer certaines bibliothèques, Composer utilise Git.

Installer Git

Comme on l'a vu, pour récupérer les bibliothèques, Composer se base sur les informations répertoriées sur Packagist. Si pour certaines bibliothèques Composer peut télécharger directement des archives contenant les sources, pour d'autres il doit utiliser un gestionnaire de versions tel que Git.

En réalité, beaucoup de bibliothèques sont dans ce cas, c'est pourquoi l'installation de Git ne peut être évitée. Ce n'est pas grave, attaquons-la ! Je vais ici décrire rapidement son installation, mais sachez qu'[un cours du Site du Zéro](#) détaille très bien son fonctionnement et son installation.

Installer Git sous Windows

Sous Windows, il faut utiliser [msysgit](#). Cela installe msys (un système d'émulation des commandes Unix sous Windows) et Git lui-même.

Téléchargez le fichier et exécutez-le, cela va tout installer. Laissez les paramètres par défaut, ils conviennent très bien. Cela va prendre un peu de temps, car il y a pas mal à télécharger (une centaine de Mo) et à exécuter, ne vous inquiétez pas. Une fois que vous avez une ligne de commande (`/dev`), vous pouvez fermer la fenêtre.

Une fois cela installé, il faut ajouter les exécutables Git au PATH de Windows. Ajoutez donc ceci : «
;`C:\msysgit\bin;C:\msysgit\mingw\bin` » à la suite de votre variable d'environnement système PATH (on en a déjà parlé dans le [premier chapitre](#)).

Redémarrez votre ordinateur, et ensuite vérifiez l'installation en exécutant la commande suivante :

Code : Console

```
C:\wamp\www>git version
git version 1.7.9.msysgit.0
```

Si vous n'avez pas d'erreur, c'est tout bon !

Installer Git sous Linux

Sous Linux, c'est encore plus simple avec votre gestionnaire de paquets. Voici comment l'installer depuis la distribution Debian et ses dérivées (Ubuntu, etc.) :

Code : Console

```
sudo apt-get install git-core
```

**Installer un bundle grâce à Composer
Manipulons Composer**

Avant d'utiliser Composer dans notre projet Symfony2, on va d'abord s'amuser avec lui sur un projet test afin de bien comprendre son fonctionnement. Créez donc un répertoire `test` là où vous avez téléchargé Composer.

Déclarer ses dépendances

La première chose à faire dans un projet, c'est de déclarer ses dépendances. Cela se fait via un fichier `composer.json`, qui contient les informations sur les bibliothèques dont dépend votre projet ainsi que leur version. La syntaxe est assez simple, en JSON, créez le fichier suivant :

Code : JavaScript

```
// test/composer.json

{
    "require": {
        "twig/extensions": "dev-master"
    }
}
```

Ce tableau JSON est le minimum syndical : il ne précise que les dépendances via la clé `require`. Il n'y a ici qu'une seule dépendance : `"twig/extensions"`. La version requise pour cette dépendance est `"dev-master"`, cela signifie qu'on veut la version la plus à jour possible.

Un point sur les versions, voici ce que vous pouvez mettre :

Valeur	Exemple	Description
Un numéro de version exact	<code>"2.0.17"</code>	Ainsi, Composer téléchargerera cette version exacte.
Une plage de versions	<code>">=2.0,<3.0"</code>	Ainsi, Composer téléchargerera la version la plus à jour, à partir de la version 2.0 et en s'arrêtant avant la version 3.0. Par exemple, si les dernières versions sont 2.9, 3.0 et 3.1, Composer téléchargerera la version 2.9.
Un numéro de version avec joker « * »	<code>"2.0.*"</code>	Ainsi, Composer téléchargerera la version la plus à jour qui commence par 2.0. Par exemple, il téléchargerait la version 2.0.17, mais pas la version 2.1.1. C'est la façon la plus utilisée pour définir la version des dépendances.

Dans notre cas, "dev-master" correspond à un numéro de version exact : le dernier disponible. C'est un cas particulier.

Pour information, vous pouvez aller regarder les informations de cette bibliothèque sur [Packagist](#). Vous pouvez voir qu'elle dépend d'une autre bibliothèque, "twig/twig", qui correspond au moteur de templates Twig à proprement parler. Il en a besoin dans sa version "1.*", Composer ira donc chercher la dernière version dans la branche 1.*.

Mettre à jour les dépendances

Pour mettre à jour toutes les dépendances, "twig/extensions" dans notre cas, il faut exécuter la commande `update` de Composer, comme ceci :

Code : Console

```
C:\wamp\www\test>php ./composer.phar update
Loading composer repositories with package information
Updating dependencies
  - Installing twig/twig (v1.10.0)
    Downloading: 100%
  - Installing twig/extensions (dev-master dcdf02)
    Cloning dcdf02fbac1282e6b8f4d0558cc7e9580105688

Writing lock file
Generating autoload files
```



Chez moi, j'ai placé Composer (le fichier `composer.phar`) dans le répertoire `www`. Or ici on travaille dans le répertoire de test `www\test`. J'ai donc dit à PHP d'exécuter le fichier `./composer.phar`, mais bien sûr si le vôtre est dans le répertoire courant ou ailleurs, adaptez la commande. 😊

Et voilà !

Vous pouvez aller vérifier dans le répertoire `test/vendor` :

- Composer a téléchargé la dépendance "twig/extensions" que l'on a défini, dans `vendor/twig/extensions` ;
- Composer a téléchargé la dépendance "twig/twig" de notre dépendance à nous, dans `vendor/twig/twig` ;
- Composer a généré les fichiers nécessaires pour l'`autoload`, allez vérifier le fichier `vendor/composer/autoload_namespaces.php`.

Tout est maintenant OK pour se servir de "twig/extensions" dans votre projet ! C'était donc la démarche et le fonctionnement pour la gestion des dépendances avec Composer. Mais revenons maintenant à notre projet sous Symfony2.

Mettons à jour Symfony2



Ce paragraphe s'adresse à ceux qui ont téléchargé l'archive « with vendors » de Symfony2, et qui ne l'ont pas déjà téléchargée via Composer.

Si vous avez téléchargé la version de Symfony2 qui comprend déjà les dépendances, vous avez en fait téléchargé tout le contenu du dossier `vendor` que Composer pourrait gérer tout seul. L'objectif de ce paragraphe est de déléguer cette gestion à Composer.

Vous pouvez voir qu'en fait il existe déjà un fichier de définition des dépendances à la racine de votre projet : le fichier `composer.json`. N'hésitez pas à l'ouvrir : vous pourrez y voir toutes les dépendances déjà définies.

Pour l'instant, ce fichier existe, mais on n'a jamais utilisé Composer pour les gérer. Il ne reste donc plus qu'à dire à Composer de

les mettre toutes à jour. Rien de spécial à faire par rapport à tout à l'heure, exécutez simplement la commande suivante :

Code : Console

```
php ./composer.phar update
```

Cela va prendre un peu de temps, car Composer a beaucoup à télécharger, les dépendances d'un projet Symfony2 sont nombreuses. Il y a en effet Symfony2 en lui-même, mais également Doctrine, Twig, certains bundles, etc.

Maintenant, Composer a pris la main sur toutes vos dépendances, on va pouvoir en ajouter une nouvelle : un bundle Symfony2 !

Installer un bundle avec Composer

Dans ce paragraphe, nous allons installer le bundle DoctrineFixtureBundle, qui permet de préremplir la base de données avec des données, afin de bien tester votre application. Cependant, les explications sont valables pour l'installation de n'importe quel bundle, retenez donc bien la méthode.

1. Trouver le nom du bundle

Vous l'avez compris, on définit une dépendance dans Composer via son nom. Il faut donc logiquement connaître ce nom pour pouvoir l'installer. Pour cela, rien de plus simple, on se rend sur <http://packagist.org/> et on fait une petite recherche. Dans notre cas, recherchez « fixture », et cliquez sur le bundle de Doctrine, « doctrine/doctrine-fixtures-bundle ».

2. Déterminer la version du bundle

Une fois que vous avez trouvé votre bundle, il faut en sélectionner une version. Il se peut que celui-ci n'ait pas vraiment de version fixe, et que seul "dev-master" soit disponible. Dans ce cas, assurez-vous (auprès du développeur, ou en regardant le code) qu'il est compatible avec votre projet.

Mais la plupart du temps, les bundles sont versionnés et c'est à vous de choisir la version qui vous convient. Restons sur notre cas du bundle fixture : <https://packagist.org/packages/doctrine/doctrine-fixtures-bundle>. Les deux dernières versions sont "dev-master" et "2.0.x-dev" :

- Regardez les prérequis de la version 2.0.x-dev : il est indiqué que cette version a besoin de "symfony/symfony" dans sa version 2.0 (car "<2.1" exclut la version 2.1). Cette version est trop vieille, on ne peut donc pas l'utiliser, car on tourne sur un Symfony 2.2 ;
- Regardez alors les prérequis de la version dev-master : on a besoin de "doctrine/doctrine-bundle" (cliquez dessus), qui lui-même a besoin de "symfony/framework-bundle" dans sa version 2.1 ou 2.2 ("<2.3" exclut la 2.3, rappelez-vous). Ce bundle suit la même numérotation de version que le framework Symfony2, la version 2.2 est donc OK pour nous.

On choisit alors la version "dev-master" du bundle.

3. Déclarer le bundle à Composer

Une fois qu'on a le nom du bundle et sa version, il faut le déclarer à Composer, dans le fichier `composer.json`. On sait déjà le faire, il faut modifier la section "require", voici ce que cela donne :

Code : JavaScript

```
// composer.json  
// ...  
"require": {  
    "php": ">=5.3.3",  
    // ...  
    "jms/di-extra-bundle": "1.1.*",
```

```
"doctrine/doctrine-fixtures-bundle": "dev-master",
"doctrine/data-fixtures": "@dev"
},
// ...
```

Nous avons affaire à un cas particulier. Notre Composer est paramétré pour ne prendre que des versions stables, sauf si on le lui dit explicitement. Avec twig/extensions et maintenant doctrine/doctrine-fixtures-bundle, on a explicitement précisé qu'on voulait une version de développement ("dev-master"). Or, si tout à l'heure toutes les dépendances de twig/extensions existaient en version stable, ce n'est pas le cas de doctrine/doctrine-fixtures-bundle : la dépendance doctrine/data-fixtures n'existe pas encore en version stable. On est donc obligé de dire à Composer qu'on accepte les versions de développement de cette dépendance : d'où la ligne doctrine/data-fixtures: "@dev". Le "@dev" signifie qu'on ne précise pas de numéro de version exacte, mais qu'on accepte également les versions de développement.



N'oubliez pas d'ajouter une virgule à la fin de l'avant-dernière dépendance, dans mon cas "jms/di-extra-bundle" !

4. Mettre à jour les dépendances

Une fois la dépendance déclarée à Composer, il ne reste qu'à mettre à jour les dépendances, avec la commande `update` :

Code : Console

```
C:\wamp\www\Symfony>php ../composer.phar update
Loading composer repositories with package information
Updating dependencies
- Installing doctrine/data-fixtures (dev-master a95d783)
  Cloning a95d7839a7794c7c9b22d64e859ee70658d977fe

- Installing doctrine/doctrine-fixtures-bundle (dev-master 9edc67a)
  Cloning 9edc67af16e736a31605e7fa9c9e3edbd9db6427

Writing lock file
Generating autoload files
Clearing the cache for the dev environment with debug true
Installing assets using the hard copy option
[...]
```

5. Enregistrer le bundle dans le Kernel

Dernier point, totalement indépendant de Composer : il faut déclarer le bundle dans le Kernel de Symfony2. Allez dans app/AppKernel.php et ajoutez la ligne 8 :

Code : PHP

```
<?php
// app/AppKernel.php

// ...

if (in_array($this->getEnvironment(), array('dev', 'test'))) {
    // ...
    $bundles[] = new
        Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle();
}
```

```
// ...
```



Ici, j'ai déclaré le bundle uniquement pour les modes « dev » et « test », car c'est l'utilité du bundle fixture, on en reparlera. Bien entendu, si votre bundle doit être accessible en mode « prod », placez-le en conséquence. 😊

Voilà, votre bundle est opérationnel !

Attention si vous avez l'habitude de Symfony2.0 où il fallait également déclarer le namespace pour l'*autoload* : perdez cette habitude tout de suite ! Composer s'occupe vraiment de tout, et notamment de déclarer les namespaces pour l'*autoload* : allez le vérifier dans le fichier `vendor/composer/autoload_namespaces.php`. Ce fichier contient tous les namespaces nécessaires pour votre projet, et Symfony2 l'inclut déjà, vérifiez-le en regardant le fichier `app/autoload.php`, on y voit cette ligne :

Code : PHP

```
<?php
if (!$_loader = @include __DIR__. '/../../vendor/autoload.php') {
// ...
```

Voilà comment Symfony2 utilise Composer pour gérer son *autoload*.

Gérer l'*autoload* d'une bibliothèque manuellement

Il se peut que vous ayez une bibliothèque existante en stock, mais qui n'est pas référencée sur Packagist. Composer ne peut pas gérer entièrement cette bibliothèque, car il n'a pas ses informations : comment la mettre à jour, quelles sont ses dépendances, etc.

Par contre, vous avez tout de même envie de l'intégrer dans votre projet. Notamment, vous souhaitez la charger automatiquement grâce à l'*autoload* PHP. Pour ce faire, il faut ajouter la section "autoload" à votre `composer.json`, dans laquelle Composer ne mettra pas son nez pour tout ce qui est installation et mises à jour. Par contre, il l'inclura dans son fichier *autoload* que Symfony2 charge. Voici ce que vous devez rajouter :

Code : JavaScript

```
// composer.json

{
// ...

"autoload": {
    "psr-0": {
        "VotreNamespace": "chemin/vers/la/bibliothèque"
    }
}

// ...
}
```



Attention, il faut toujours utiliser cette méthode et ne jamais aller modifier le fichier `vendor/composer/autoload_namespaces.php` ! Comme tout fichier qui se trouve dans le répertoire `vendor`, vous ne devez pas le toucher, car il peut être écrasé à tout moment : dès que vous faites un `update` avec Composer, ce dernier va télécharger les nouvelles versions et écraser les anciennes...

Bien sûr, pour que cela fonctionne il faut que votre bibliothèque respecte la convention [PSR-0](#), c'est-à-dire une convention de nommage et d'*autoload*. Je vous invite à lire le lien pour en savoir plus à ce propos. Symfony2 suit bien entendu cette convention.

Pour conclure

Ce chapitre-parentthèse sur Composer touche à sa fin. S'il vous semble un peu décalé aujourd'hui, vous me remercierez un peu plus tard de vous en avoir parlé, lorsque vous voudrez installer des bundles trouvés à droite ou à gauche. D'ailleurs, on a déjà installé DoctrineFixtureBundle, un bundle bien pratique dont nous nous resservirons dès la prochaine partie sur Doctrine !

Sachez également que je n'ai absolument pas tout dit sur Composer, car cela ferait trop long et ce n'est pas tellement l'objet de ce tutoriel. Cependant, Composer a sa propre documentation et je vous invite à vous y référer. Par curiosité, par intérêt, en cas de problème, n'hésitez pas : <http://getcomposer.org> !

En résumé

- Composer est un outil pour gérer les dépendances d'un projet en PHP, qu'il soit sous Symfony2 ou non.
- Le fichier `composer.json` permet de lister les dépendances que doit inclure Composer dans votre projet.
- Composer détermine la meilleure version possible pour vos dépendances, les télécharge, et configure leur *autoload* tout seul.
- Composer trouve toutes les bibliothèques sur le site <http://www.packagist.org>, sur lequel vous pouvez envoyer votre propre bibliothèque si vous le souhaitez.
- La très grande majorité des bundles Symfony2 sont installables avec Composer, ce qui simplifie énormément leur utilisation dans un projet.

Les services, théorie et création

Vous avez souvent eu besoin d'exécuter une certaine fonction à plusieurs endroits différents dans votre code ? Ou de vérifier une condition sur toutes les pages ? Alors ce chapitre est fait pour vous ! Nous allons découvrir ici une fonctionnalité importante de Symfony : le système de services. Vous le verrez, les services sont utilisés partout dans Symfony2, et sont une fonctionnalité incontournable pour commencer à développer sérieusement un site internet sous Symfony2.

Ce chapitre ne présente que des notions sur les services, juste ce qu'il vous faut savoir pour les manipuler simplement. Nous verrons dans un prochain chapitre leur utilisation plus poussée.

C'est parti !

Pourquoi utiliser des services ?

Genèse

Vous l'avez vu jusqu'ici, une application PHP, qu'elle soit faite avec Symfony2 ou non, utilise beaucoup d'objets PHP. Un objet remplit une fonction comme envoyer un e-mail, enregistrer des informations dans une base de données, etc. Vous pouvez créer vos propres objets qui auront les fonctions que vous leur donnez. Bref, une application est en réalité un moyen de faire travailler tous ces objets ensemble, et de profiter du meilleur de chacun d'entre eux.

Dans bien des cas, un objet a besoin d'un ou plusieurs autres objets pour réaliser sa fonction. Se pose alors la question de savoir comment organiser l'instanciation de tous ces objets. Si chaque objet a besoin d'autres objets, par lequel commencer ?

L'objectif de ce chapitre est de vous présenter le conteneur de services. Chaque objet est défini en tant que service, et le conteneur de services permet d'instancier, d'organiser et de récupérer les nombreux services de votre application. Étant donné que tous les objets fondamentaux de Symfony2 utilisent le conteneur de services, nous allons apprendre à nous en servir. C'est une des fonctionnalités incontournables de Symfony2, et c'est ce qui fait sa très grande flexibilité.

Qu'est-ce qu'un service ?

Un service est simplement un objet PHP qui remplit une fonction, associé à une configuration.

Cette fonction peut être simple : envoyer des e-mails, vérifier qu'un texte n'est pas un spam, etc. Mais elle peut aussi être bien plus complexe : gérer une base de données (le service Doctrine !), etc.

Un service est donc un objet PHP qui a pour vocation d'être accessible depuis n'importe où dans votre code. Pour chaque fonctionnalité dont vous aurez besoin dans toute votre application, vous pourrez créer un ou plusieurs services (et donc une ou plusieurs classes et leur configuration). Il faut vraiment bien comprendre cela : un service est avant tout une *simple classe*.

Quant à la configuration d'un service, c'est juste un moyen de l'enregistrer dans le conteneur de services. On lui donne un nom, on précise quelle est sa classe, et ainsi le conteneur a la carte d'identité du service.

Prenons pour exemple l'envoi d'e-mails. On pourrait créer une classe avec comme nom `MonMailer` et la définir comme un service grâce à un peu de configuration. Elle deviendrait alors utilisable n'importe où grâce au conteneur de services. En réalité, Symfony2 intègre déjà une classe nommée `Swift_Mailer`, qui est enregistrée en tant que service `Mailer` via sa configuration.

Pour ceux qui connaissent, le concept de service est un bon moyen d'éviter d'utiliser trop souvent à mauvais escient le pattern singleton (utiliser une méthode statique pour récupérer l'objet depuis n'importe où).

L'avantage de la programmation orientée services

L'avantage de réfléchir sur les services est que cela force à bien séparer chaque fonctionnalité de l'application. Comme chaque service ne remplit qu'une seule et unique fonction, ils sont facilement réutilisables. Et vous pouvez surtout facilement les développer, les tester et les configurer puisqu'ils sont assez indépendants. Cette façon de programmer est connue sous le nom d'architecture orientée services, et n'est pas unique à Symfony2 ni au PHP.

Le conteneur de services

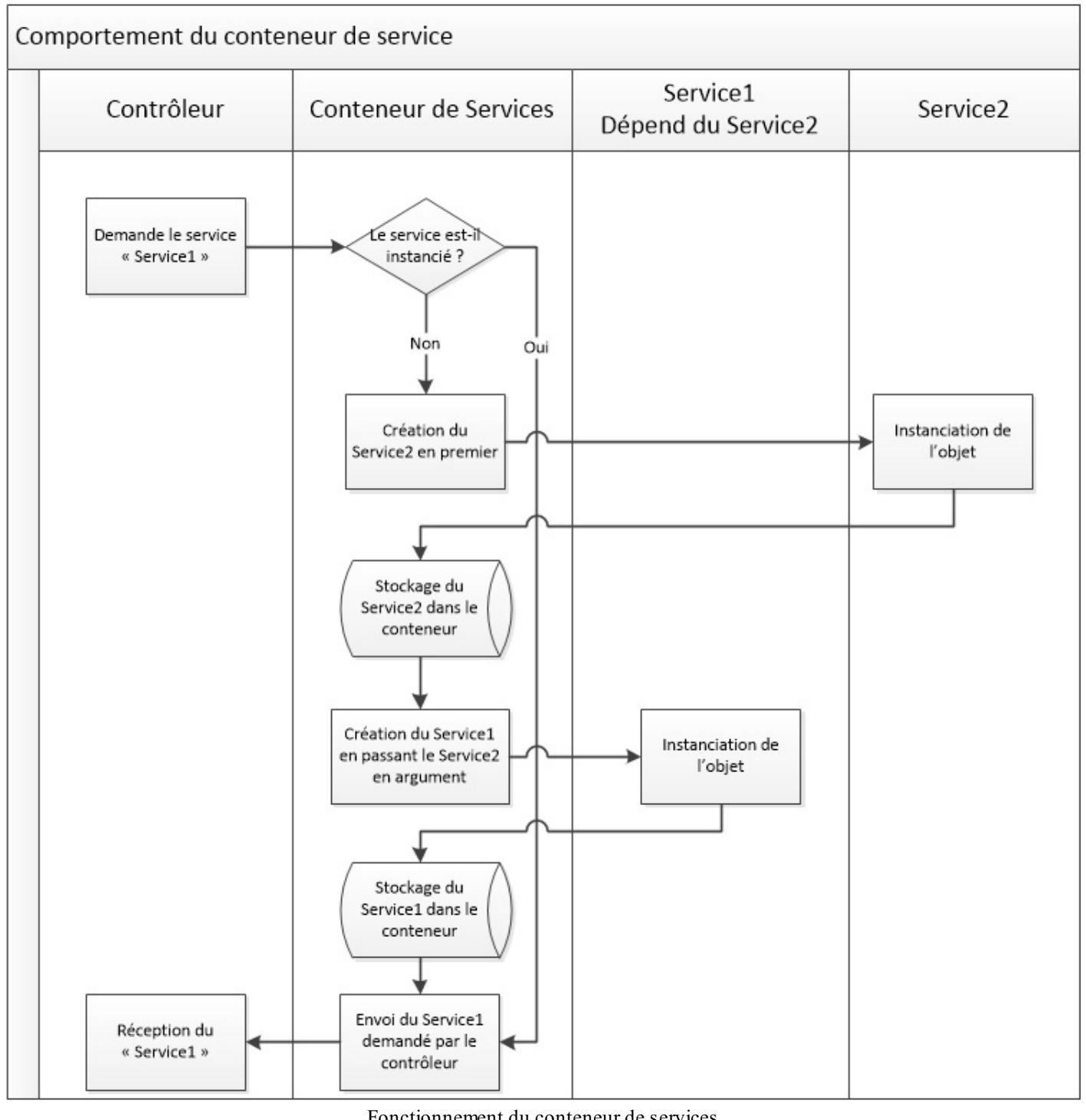


Mais alors, si un service est juste une classe, pourquoi appeler celle-ci un service ? Et pourquoi utiliser les services ?

L'intérêt réel des services réside dans leur association avec le conteneur de services. Ce conteneur de services (*services container* en anglais) est une sorte de super-objet qui gère tous les services. Ainsi, pour accéder à un service, il faut passer par le conteneur.

L'intérêt principal du conteneur est d'organiser et d'instancier (créer) vos services très facilement. L'objectif est de simplifier au maximum la récupération des services depuis votre code à vous (depuis le contrôleur ou autre). Vous demandez au conteneur un certain service en l'appelant par son nom, et le conteneur s'occupe de tout pour vous retourner le service demandé.

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



Vous voyez que le conteneur de services fait un grand travail, mais que son utilisation (depuis le contrôleur) est vraiment simple.

Comment définir les dépendances entre services ?

Maintenant que vous concevez le fonctionnement du conteneur, il faut passer à la configuration des services. Comment dire au conteneur que le `Service2` doit être instancié avant le `Service1`? Cela se fait grâce à la configuration dans Symfony2.

L'idée est juste de définir pour chaque service :

- Son nom, qui permettra de l'identifier au sein du conteneur ;
- Sa classe, qui permettra au conteneur d'instancier le service ;
- Les arguments dont il a besoin. Un argument peut être un autre service, mais aussi un paramètre (défini dans le fichier `parameters.yml` par exemple).

Nous allons voir la syntaxe de la configuration d'ici peu.

La persistance des services

Il reste un dernier point à savoir avant d'attaquer la pratique. Dans Symfony2, chaque service est « *persistant* ». Cela signifie simplement que la classe du service est instanciée une seule fois (à la première récupération du service). Si un nouvel appel est fait du même service, c'est cette instance qui est retournée et donc utilisée par la suite. Cette persistance permet de manipuler très facilement les services tout au long du processus. Concrètement, c'est le même objet `$service1` qui sera utilisé dans toute votre application.

Utiliser un service en pratique

Récupérer un service

Continuons sur notre exemple d'e-mail. Comme je vous l'ai mentionné, il existe dans Symfony un composant appelé `Swiftmailer`, qui permet d'envoyer des e-mails simplement. Il est présent par défaut dans Symfony, sous forme du service `Mailer`. Ce service est déjà créé, et sa configuration est déjà faite, il ne reste plus qu'à l'utiliser !

Pour accéder à un service déjà enregistré, il suffit d'utiliser la méthode `get ($nomDuService)` du conteneur. Par exemple :

Code : PHP

```
<?php  
$container->get('mailer');
```



Pour avoir la liste des services disponibles, utilisez la commande `php app/console container:debug`.



Et comment j'accède à `$container`, moi ?!

En effet, la question est importante. Dans Symfony, il existe une classe nommée `ContainerAware` qui possède un attribut `$container`. Le cœur de Symfony alimente ainsi les classes du framework en utilisant la méthode `setContainer ()`. Donc pour toute classe de Symfony héritant de `ContainerAware`, on peut faire ceci :

Code : PHP

```
<?php  
$this->container->get('mailer');
```

Heureusement pour nous, la classe de base des contrôleurs nommée `Controller` hérite de cette classe `ContainerAware`, on peut donc appliquer ceci aux contrôleurs :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller; // Cette
classe étend ContainerAware

class BlogController extends Controller
{
    public function indexAction()
    {
        $mailer = $this->container->get('mailer'); // On a donc accès au
conteneur

        // On peut envoyer des e-mails, etc.
    }
}
```

 Il se peut que vous ayez déjà rencontré, depuis un contrôleur, l'utilisation de `$this->get()` sans passer par l'attribut `$container`. C'est parce que la classe `Controller` fournit un raccourci, la méthode `$this->get()` faisant simplement appel à la méthode `$this->container->get()`. Donc dans un contrôleur, `$this->get('...')` est strictement équivalent à `$this->container->get('...')`.

Et voilà ! Vous savez utiliser le conteneur de services depuis un contrôleur, vraiment très simple comme on l'a vu précédemment.

Créer un service simple

Création de la classe du service

Maintenant que nous savons utiliser un service, apprenons à le créer. Comme un service n'est qu'une classe, il suffit de créer un fichier n'importe où et de créer une classe dedans.

La seule convention à respecter, de façon générale dans Symfony, c'est de mettre notre classe dans un namespace correspondant au dossier où est le fichier. C'est la norme [PSR-0](#) pour l'autoload. Par exemple, la classe `Sdz\BlogBundle\Antispam\SdzAntispam` doit se trouver dans le répertoire `src/Sdz/BlogBundle/Antispam/SdzAntispam.php`. C'est ce que nous faisons depuis le début du cours. 😊

Je vous propose, pour suivre notre fil rouge du blog, de créer un système anti-spam. Notre besoin : détecter les spams à partir d'un simple texte. Comme c'est une fonction à part entière, et qu'on aura besoin d'elle à plusieurs endroits (pour les articles et pour les commentaires), faisons-en un service. Ce service devra être réutilisable simplement dans d'autres projets Symfony : il ne devra pas être dépendant d'un élément de notre blog. Je nommerai ce service « `SdzAntispam` », mais vous pouvez le nommer comme vous le souhaitez. Il n'y a pas de règle précise à ce niveau, mis à part que l'utilisation des underscores (« `_` ») est fortement déconseillée.

Je mets cette classe dans le répertoire `/Antispam` de notre bundle, mais vous pouvez à vrai dire faire comme vous le souhaitez.

Créons donc le fichier `src/Sdz/BlogBundle/Antispam/SdzAntispam.php`, avec ce code pour l'instant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Antispam/SdzAntispam.php

namespace Sdz\BlogBundle\Antispam;

class SdzAntispam
{}
```

C'est tout ce qu'il faut pour avoir un service. Il n'y a vraiment rien d'obligatoire, vous y mettez ce que vous voulez. Pour l'exemple, faisons un rapide anti-spam : considérons qu'un message est un spam s'il contient au moins trois liens ou adresses e-mail. Voici ce que j'obtiens :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Antispam/SdzAntispam.php

namespace Sdz\BlogBundle\Antispam;

class SdzAntispam
{
    /**
     * Vérifie si le texte est un spam ou non
     * Un texte est considéré comme spam à partir de 3 liens
     * ou adresses e-mail dans son contenu
     *
     * @param string $text
     */
    public function isSpam($text)
    {
        return ($this->countLinks($text) + $this->countMails($text)) >=
3;
    }

    /**
     * Compte les URL de $text
     *
     * @param string $text
     */
    private function countLinks($text)
    {
        preg_match_all(
            '#(http|https|ftp)://([A-Z0-9] [A-Z0-9_-]*(:[A-Z0-9] [A-Z0-9_-])*)+:(d+)?/?#i',
            $text,
            $matches);

        return count($matches[0]);
    }

    /**
     * Compte les e-mails de $text
     *
     * @param string $text
     */
    private function countMails($text)
    {
        preg_match_all(
            '#[a-z0-9._-]+@[a-z0-9._-]{2,}.\.[a-z]{2,4}#i',
            $text,
            $matches);

        return count($matches[0]);
    }
}
```

La seule méthode publique de cette classe est `isSpam()`, c'est celle que nous utiliserons par la suite. Elle retourne `true` si le message donné en argument (variable `$text`) est identifié en tant que spam, `false` sinon.

Création de la configuration du service

Maintenant que nous avons créé notre classe, il faut la signaler au conteneur de services, c'est ce qui va en faire un service en tant que tel. Un service se définit par sa classe ainsi que sa configuration. Pour cela, nous pouvons utiliser le fichier `src/Sdz/BlogBundle/Resources/config/services.yml`.



Si vous avez généré votre bundle avec le generator en répondant « oui » pour créer toute la structure du bundle, alors ce fichier `services.yml` est chargé automatiquement. Vérifiez-le en confirmant que le répertoire `DependencyInjection` de votre bundle existe, il devrait contenir le fichier `SdzBlogExtension.php`.

Si ce n'est pas le cas, vous devez créer le fichier `DependencyInjection/SdzBlogExtension.php` (adaptez à votre bundle évidemment). Mettez-y le contenu suivant, qui permet de charger automatiquement le fichier `services.yml` que nous allons modifier :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/DependencyInjection/SdzBlogExtension.php

namespace Sdz\BlogBundle\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\Loader;

class SdzBlogExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new Loader\YamlFileLoader($container, new
FileLocator(__DIR__.'/../Resources/config'));
        $loader->load('services.yml');
    }
}
```

La méthode `load()` de cet objet est automatiquement exécutée par Symfony2 lorsque le bundle est chargé. Et dans cette méthode, on charge le fichier de configuration `services.yml`, ce qui permet d'enregistrer la définition des services qu'il contient dans le conteneur de services. Fin de la parenthèse.

Revenons à notre fichier de configuration. Ouvrez ou créez le fichier `Resources/config/services.yml` de votre bundle, et ajoutez-y la configuration pour notre service :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdz_blog_antispam:
        class: Sdz\BlogBundle\Antispam\SdzAntispam
```

Dans cette configuration :

- `sdz_blog_antispam` est le nom de notre service fraîchement créé. De cette manière, le service sera accessible via `<?php $container->get('sdz_blog_antispam');`. Essayez de respecter la convention en préfixant le nom de vos services par « nomApplication_nomBundle ». Pour notre bundle `Sdz\BlogBundle`, on a donc préfixé notre service de « `sdz_blog` ».
- `class` est un attribut obligatoire de notre service, il définit simplement le namespace complet de la classe du service. Cela permet au conteneur de services d'instancier la classe lorsqu'on le lui demandera.

Il existe bien sûr d'autres attributs pour affiner la définition de notre service, nous les verrons dans le prochain chapitre sur les services.

Sachez également que le conteneur de Symfony2 permet de stocker aussi bien des services (des classes) que des paramètres (des variables). Pour définir un paramètre, la technique est la même que pour un service, dans le fichier `services.yml` :

Code : YAML

```
parameters:  
    mon_parametre: ma_valeur  
  
services:  
    # ...
```

Et pour accéder à ce paramètre, la technique est la même également, sauf qu'il faut utiliser la méthode `<?php $container->getParameter('nomParametre');` au lieu de `get()`.

Utilisation du service

Maintenant que notre classe est définie, et notre configuration déclarée, nous avons affaire à un vrai service. Voici un exemple simple de l'utilisation que l'on pourrait en faire :

Code : PHP

```
<?php  
// src/Sdz/BlogBundle/Controller/BlogController.php  
  
namespace Sdz\BlogBundle\Controller;  
  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
  
class BlogController extends Controller  
{  
    public function indexAction()  
    {  
        // On récupère le service  
        $antispam = $this->container->get('sdz_blog.antispam');  
  
        // Je pars du principe que $text contient le texte d'un message  
        // quelconque  
        if ($antispam->isSpam($text)) {  
            throw new \Exception('Votre message a été détecté comme spam  
!');  
        }  
  
        // Le message n'est pas un spam, on continue l'action...  
    }  
}
```

Et voilà, vous avez créé et utilisé votre premier service !

Si vous définissez la variable `$text` avec 3 adresses e-mail, vous aurez droit au message d'erreur de la figure suivante.



Mon message

était du spam

Créer un service avec des arguments

Passer à la vitesse supérieure

Nous avons un service flambant neuf et opérationnel. Parfait. Mais on n'a pas utilisé toute la puissance du conteneur de services que je vous ai promise : l'utilisation interconnectée des services.

Injecter des arguments dans nos services

En effet, la plupart du temps vos services ne fonctionneront pas seuls, et vont nécessiter l'utilisation d'autres services, de paramètres ou de variables. Il a donc fallu trouver un moyen propre et efficace pour pallier ce problème, et c'est le conteneur de services qui propose la solution ! Pour passer des arguments à votre service, il faut utiliser la configuration du service :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdz_blog_antispam:
        class: Sdz\BlogBundle\Antispam\SdzAntispam
        arguments: [] # Tableau d'arguments
```

Les arguments peuvent être :

- Des valeurs normales en YAML (des booléens, des chaînes de caractères, des nombres, etc.) ;
- Des paramètres (définis dans le `parameters.yml` par exemple) : l'identifiant du paramètre est encadré de signes « % » : `%nomDuParametre%` ;
- Des services : l'identifiant du service est précédé d'une arobase : `@nomDuService`.

Pour faire l'utilisation de ces trois types d'arguments, je vous propose d'injecter différentes valeurs dans notre service d'antispam, comme ceci par exemple :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdz_blog_antispam:
        class: Sdz\BlogBundle\Antispam\SdzAntispam
        arguments: [@mailer, %locale%, 3]
```

Dans cet exemple, notre service utilise :

- `@mailer` : le service Mailer (pour envoyer des e-mails) ;

- %locale% : le paramètre locale (pour récupérer la langue) :
- 3 : et le nombre 3 (qu'importe son utilité !).

Une fois vos arguments définis dans la configuration, il vous suffit de les récupérer avec le constructeur du service. Les arguments de la configuration et ceux du constructeur vont donc de paire. Si vous modifiez l'un, n'oubliez pas d'adapter l'autre. Voici donc le constructeur adapté à notre nouvelle configuration :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Antispam/SdzAntispam.php

namespace Sdz\BlogBundle\Antispam;

class SdzAntispam
{
    protected $mailer;
    protected $locale;
    protected $nbForSpam;

    public function __construct(\Swift_Mailer $mailer, $locale,
$nbForSpam)
    {
        $this->mailer = $mailer;
        $this->locale = $locale;
        $this->nbForSpam = (int) $nbForSpam;
    }

    /**
     * Vérifie si le texte est un spam ou non
     * Un texte est considéré comme spam à partir de 3 liens
     * ou adresses e-mails dans son contenu
     *
     * @param string $text
     */
    public function isSpam($text)
    {
        // On utilise maintenant l'argument $this->nbForSpam et non
        plus le « 3 » en dur :
        return ($this->countLinks($text) + $this->countMails($text)) >=
        $this->nbForSpam;
    }

    // ... on pourrait également utiliser $this->mailer pour prévenir
    d'un spam l'administrateur par exemple
}
```

Le constructeur est très simple, l'idée est juste de récupérer les arguments pour les stocker dans les attributs de la classe. L'ordre des arguments du constructeur est le même que l'ordre des arguments définis dans la configuration du service.

Vous pouvez voir que j'ai également modifié la méthode `isSpam()` pour vous montrer comment utiliser un argument. Ici, j'ai remplacé le « 3 » que j'avais mis en dur précédemment par la valeur de l'argument `nbForSpam`. Ainsi, si vous décidez de passer cette valeur à 10 au lieu de 3, vous ne modifiez que la configuration du service, sans toucher à son code !

L'injection de dépendances

Vous ne vous en êtes pas forcément aperçus, mais on vient de réaliser quelque chose d'assez exceptionnel ! En une seule ligne de configuration, on vient d'injecter un service dans un autre. Ce mécanisme s'appelle l'injection de dépendances (*dependency injection* en anglais).

L'idée, comme on l'a vu précédemment, c'est que le conteneur de services s'occupe de tout. Votre service a besoin du service Mailer ? Pas de soucis, précisez-le dans sa configuration, et le conteneur de services va prendre soin d'instancier Mailer, puis de vous le transmettre à l'instanciation de votre service.

Cela permet à un service d'utiliser d'autres services. En fait, on pourrait également injecter tout le conteneur dans un service, qui aurait ainsi accès à tous les autres services, au même titre que les contrôleurs. Mais ce n'est pas très propre, et il faut essayer de bien découpler vos services et surtout leurs dépendances.

Vous pouvez bien entendu utiliser votre nouveau service dans un prochain service. Au même titre que vous avez mis @mailer en argument, vous pourrez mettre @sdz_blog.antispam ! 😊

Ainsi retenez bien : lorsque vous développez un service dans lequel vous auriez besoin d'un autre, injectez-le dans les arguments de la configuration, et libérez la puissance de Symfony2 !

Pour conclure

Je me permets d'insister sur un point : les services et leur conteneur sont l'élément crucial et inévitable de Symfony2. Les services sont utilisés intensément par le cœur même du framework, et nous serons amenés à en créer assez souvent dans la suite de ce cours.

Gardez en tête que leur intérêt principal est de bien découpler les fonctions de votre application. Tout ce que vous comptez utiliser à plusieurs endroits dans votre code mérite un service. Gardez vos contrôleurs les plus simples possible, et n'hésitez pas à créer des services qui contiennent la logique de votre application. 😊

Ce chapitre vous a donc apporté les connaissances nécessaires pour définir et utiliser simplement les services. Bien sûr, il y a bien d'autres notions à voir, mais nous les verrons un peu plus loin dans [un prochain chapitre](#).

Si vous souhaitez aborder plus en profondeur les notions théoriques abordées dans ce chapitre, je vous propose les lectures suivantes :

- [Introduction à l'injection de dépendances en PHP \(Site du Zéro\)](#) de vincent1870 ;
- [Les design patterns : l'injection de dépendances \(Site du Zéro\)](#) de vyk12 ;
- [Architecture orientée services \(Wikipédia\)](#).

En attendant, la prochaine partie abordera la gestion de la base de données !

En résumé

- Un service est une simple classe associée à une certaine configuration.
- Le conteneur de services organise et instancie tous vos services, grâce à leur configuration.
- Les services sont la base de Symfony2, et sont très utilisés par le cœur même du framework.
- L'injection de dépendances est assurée par le conteneur, qui connaît les arguments dont a besoin un service pour fonctionner, et les lui donne donc à sa création.

Partie 3 : Gérer la base de données avec Doctrine2

Symfony2 est livré par défaut avec l'ORM Doctrine2. Qu'est-ce qu'un ORM ? Qu'est-ce que Doctrine2 ? Ce tutoriel pour débutants est fait pour vous, car c'est ce que nous allons apprendre dans cette partie !

Lisez bien l'ensemble des chapitres de cette partie : ils forment un tout, et toutes vos questions seront résolues à la fin de la partie !

La couche métier : les entités

L'objectif d'un ORM (pour *Object-Relation Mapper*, soit en français « lien objet-relation ») est simple : se charger de l'enregistrement de vos données en vous faisant oublier que vous avez une base de données. Comment ? En s'occupant de tout ! Nous n'allons plus écrire de requêtes, ni créer de tables via phpMyAdmin. Dans notre code PHP, nous allons faire appel à Doctrine2, l'ORM par défaut de Symfony2, pour faire tout cela.

Notions d'ORM : fini les requêtes, utilisons des objets

Je vous propose de commencer par un exemple pour bien comprendre. Supposons que vous disposez d'une variable `<?php $utilisateur`, un objet `User` qui représente l'un de vos utilisateurs qui vient de s'inscrire sur votre site. Pour sauvegarder cet objet, vous êtes habitués à créer votre propre fonction qui effectue une requête SQL du type `INSERT INTO` dans la bonne table, etc. Bref, vous devez gérer tout ce qui touche à l'enregistrement en base de données. En utilisant un ORM, vous n'aurez plus qu'à utiliser quelques fonctions de cet ORM, par exemple : `<?php $orm->save($utilisateur)`. Et ce dernier s'occupera de tout ! Vous avez enregistré votre utilisateur en une seule ligne. 😊 Bien sûr, ce n'est qu'un exemple, nous verrons les détails pratiques dans la suite de ce chapitre, mais retenez bien l'idée.

Mais l'effort que vous devrez faire pour bien utiliser un ORM, c'est d'oublier votre côté « administrateur de base de données ». Oubliez les requêtes SQL, pensez objet !

Vos données sont des objets

Dans ORM, il y a la lettre O comme Objet. En effet, pour que tout le monde se comprenne, toutes vos données doivent être sous forme d'objets. Concrètement, qu'est-ce que cela implique dans notre code ? Pour reprendre l'exemple de notre utilisateur, quand vous étiez petits, vous utilisiez sûrement un tableau, puis vous accédiez à vos attributs via `<?php $utilisateur['pseudo']` ou `<?php $utilisateur['email']` par exemple. Soit, c'était très courageux de votre part. Mais nous allons aller plus loin, maintenant.

Utiliser des objets n'est pas une grande révolution en soi. Faire `<?php $utilisateur->getPseudo()` au lieu de `<?php $utilisateur['pseudo']`, c'est joli, mais limité. Ce qui est une révolution, c'est de coupler cette représentation objet avec l'ORM. Qu'est-ce que vous pensez d'un `<?php $utilisateur->getCommentaires()` ? Ha ha ! vous ne pouviez pas faire cela avec votre tableau ! Ici, la méthode `<?php $utilisateur->getCommentaires()` déclencherait la bonne requête, récupérerait tous les commentaires postés par votre utilisateur, et vous retournerait une sorte de tableau d'objets de type `Commentaire` que vous pourriez afficher sur la page de profil de votre utilisateur, par exemple. Cela commence à devenir intéressant, n'est-ce pas ?

Au niveau du vocabulaire, un objet dont vous confiez l'enregistrement à l'ORM s'appelle une **entité** (*entity* en anglais). On dit également **persister** une entité, plutôt qu'enregistrer une entité. Vous savez, l'informatique et le jargon... 😊

Créer une première entité avec Doctrine2

Une entité, c'est juste un objet

Derrière ce titre se cache la vérité. Une entité, ce que l'ORM va manipuler et enregistrer dans la base de données, ce n'est vraiment rien d'autre qu'un simple objet. Voici ce à quoi pourrait ressembler l'objet `Article` de notre blog :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

class Article
{
    protected $id;
```

```
protected $date;  
  
protected $titre;  
  
protected $auteur;  
  
protected $contenu;  
  
// Et bien sûr les getters/setters :  
  
public function setId($id)  
{  
    $this->id = $id;  
}  
public function getId()  
{  
    return $this->id;  
}  
  
public function setDate($date)  
{  
    $this->date = $date;  
}  
public function getDate()  
{  
    return $this->date;  
}  
  
public function setTitre($titre)  
{  
    $this->titre = $titre;  
}  
public function getTitre()  
{  
    return $this->titre;  
}  
  
public function setAuteur($auteur)  
{  
    $this->auteur = $auteur;  
}  
public function getAuteur()  
{  
    return $this->auteur;  
}  
  
public function setContenu($contenu)  
{  
    $this->contenu = $contenu;  
}  
public function getContenu()  
{  
    return $this->contenu;  
}
```



Inutile de créer ce fichier pour l'instant, nous allons le générer plus bas, patience. 😊

Comme vous pouvez le voir, c'est très simple. Un objet, des propriétés, et bien sûr, les *getters/setters* correspondants. On pourrait en réalité utiliser notre objet dès maintenant !

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Sdz\BlogBundle\Entity\Article;

// ...

public function testAction()
{
    $article = new Article;
    $article->setDate(new \Datetime()); // date d'aujourd'hui
    $article->setTitre('Mon dernier weekend');
    $article->setAuteur('Bibi');
    $article->setContenu("C'était vraiment super et on s'est bien
    amusé.");

    return $this->render('SdzBlogBundle:Article:test.html.twig',
array('article' => $article));
}

```

Ajoutez à cela la vue correspondante qui afficherait l'article passé en argument avec un joli code HTML, et vous avez un code opérationnel. Bien sûr, il est un peu limité car statique, mais l'idée est là et vous voyez comment l'on peut se servir d'une entité. Retenez donc : une entité n'est rien d'autre qu'un objet.

Normalement, vous devez vous poser une question : comment l'ORM va-t-il faire pour enregistrer cet objet dans la base de données s'il ne connaît rien de nos propriétés date, titre et contenu ? Comment peut-il deviner que notre propriété date doit être stockée avec un champ de type DATE dans la table ? La réponse est aussi simple que logique : il ne devine rien, on va le lui dire !

Une entité, c'est juste un objet... mais avec des commentaires !



Quoi ? Des commentaires ?

OK, je dois avouer que ce n'est pas intuitif si vous ne vous en êtes jamais servi, mais oui, on va ajouter des commentaires dans notre code et Symfony2 va se servir directement de ces commentaires pour ajouter des fonctionnalités à notre application. Ce type de commentaires se nomme **l'annotation**. Les annotations doivent respecter une syntaxe particulière, regardez par vous-mêmes :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

// On définit le namespace des annotations utilisées par Doctrine2
// En effet, il existe d'autres annotations, on le verra par la
// suite, qui utiliseront un autre namespace
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 */
class Article
{
    /**

```

```
* @ORM\Column(name="id", type="integer")
* @ORM\Id
* @ORM\GeneratedValue(strategy="AUTO")
*/
private $id;

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

/**
* @ORM\Column(name="titre", type="string", length=255)
*/
private $titre;

/**
* @ORM\Column(name="auteur", type="string", length=255)
*/
private $auteur;

/**
* @ORM\Column(name="contenu", type="text")
*/
private $contenu;

// Les getters
// Les setters
}
```



Ne recopiez toujours pas toutes ces annotations à la main, on utilise le générateur en console au paragraphe juste en dessous.



Attention par contre pour les prochaines annotations que vous serez amenés à écrire à la main : elles doivent être dans des commentaires de type « /** », avec précisément deux étoiles. Si vous essayez de les mettre dans un commentaire de type « /* » ou encore « // », elles seront simplement ignorées.

Grâce à ces annotations, Doctrine2 dispose de toutes les informations nécessaires pour utiliser notre objet, créer la table correspondante, l'enregistrer, définir un identifiant (id) en auto-incrémentation, nommer les colonnes, etc. Ces informations se nomment les *metadata* de notre entité. Je ne vais pas épiloguer sur les annotations, elles sont suffisamment claires pour être comprises par tous. 😊 Ce qu'on vient de faire, à savoir rajouter les *metadata* à notre objet Article, s'appelle *mapper* l'objet Article. C'est-à-dire faire le lien entre notre objet de base et la représentation physique qu'utilise Doctrine2.

Sachez quand même que, bien que l'on utilisera les annotations tout au long de ce tutoriel, il existe d'autres moyens de définir les *metadata* d'une entité : en YAML, en XML et en PHP. Si cela vous intéresse, vous trouverez plus d'informations sur la définition des *metadata* via les autres moyens dans le [chapitre Doctrine2 de la documentation de Symfony2](#).

Générer une entité : le générateur à la rescouisse !

En tant que bon développeurs, on est fainéants à souhait, et ça, Symfony2 l'a bien compris ! On va donc se refaire une petite session en console afin de générer notre première entité. Entrez la commande suivante et suivez le guide :

Code : Console

```
C:\wamp\www\Symfony>php app/console generate:doctrine:entity
```

1. Code : Console

```
Welcome to the Doctrine2 entity generator
```

This command helps you generate Doctrine2 entities.

First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name:_

Grâce à ce que le générateur vous dit, vous l'avez compris, il faut entrer le nom de l'entité sous le format NomBundle:NomEntité. Dans notre cas, on entre donc `SdzBlogBundle:Article`.

2.

Code : Console

```
The Entity shortcut name: SdzBlogBundle:Article
```

Determine the format to use for the mapping information.

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

Comme je vous l'ai dit, nous allons utiliser les annotations, qui sont d'ailleurs le format par défaut. Appuyez juste sur la touche Entrée.

3.

Code : Console

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

Instead of starting with a blank entity, you can add some fields now.
Note that the primary key will be added automatically (named id).

Available types: array, object, boolean, integer, smallint,
bigint, string, text, datetime, datetimetz, date, time, decimal, float,
blob.

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

On commence à saisir le nom de nos champs. Lisez bien ce qui est inscrit avant : Doctrine2 va ajouter automatiquement l'id, de ce fait, pas besoin de le redéfinir ici. On entre donc notre date : `date`.

4.

Code : Console

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

C'est maintenant que l'on va dire à Doctrine à quel type correspond notre propriété date. Voici la liste des types possibles :

array, object, boolean, integer, smallint, bigint, string, text, datetime, datetimetz, date, time, decimal, et float. Tapez donc `datetime`.

5. Répétez les points 3 et 4 pour les propriétés `titre`, `auteur` et `contenu.titre` et `auteur` sont de type `string` de 255 caractères (pourquoi pas). Contenu est par contre de type `text`.
- 6.

Code : Console

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

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

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

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

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

Lorsque vous avez fini, appuyez sur la touche Entrée.

- 7.

Code : Console

```
New field name (press <return> to stop adding fields):
Do you want to generate an empty repository class [no]? _
```

- Oui, on va créer le *repository* associé, c'est très pratique, nous en reparlerons largement. Entrez donc `yes`.
8. Confirmez la génération, et voilà !

Code : Console

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

```
Summary before generation
```

```
You are going to generate a "SdzBlogBundle:Article" Doctrine2 entity
using the "annotation" format.
```

```
Do you confirm generation [yes]? 
```

```
Entity generation
```

```
Generating the entity code: OK
```

```
You can now start using the generated code!
```

```
C:\wamp\www\Symfony>_
```

Allez tout de suite voir le résultat dans le fichier Entity/Article.php. Symfony2 a tout généré, même les *getters* et les *setters* ! Vous êtes l'heureux propriétaire d'une simple classe... avec plein d'annotations !



On a utilisé le générateur de code pour nous faciliter la vie. Mais sachez que vous pouvez tout à fait vous en passer ! Comme vous pouvez le voir, le code généré n'est pas franchement compliqué, et vous pouvez bien entendu l'écrire à la main si vous préférez.

Affiner notre entité avec de la logique métier

L'exemple de notre entité Article est un peu simple, mais rappelez-vous que la couche modèle dans une application est la couche métier. C'est-à-dire qu'en plus de gérer vos données un modèle contient également la logique de l'application. Voyez par vous-mêmes avec les exemples ci-dessous.

Attributs calculés

Prenons l'exemple d'une entité Commande, qui représenterait un ensemble de produits à acheter sur un site d'e-commerce. Cette entité aurait les attributs suivants :

- ListeProduits qui contient un tableau des produits de la commande ;
- AdresseLivraison qui contient l'adresse où expédier la commande ;
- Date qui contient la date de la prise de la commande ;
- Etc.

Ces trois attributs devront bien entendu être *mappés* (c'est-à-dire définis comme des colonnes pour l'ORM via des annotations) pour être enregistrés en base de données par Doctrine2. Mais il existe d'autres caractéristiques pour une commande, qui nécessitent un peu de calcul : le prix total, un éventuel coupon de réduction, etc. Ces caractéristiques n'ont pas à être persistées en base de données, car elles peuvent être déduites des informations que l'on a déjà. Par exemple, pour avoir le prix total, il suffit de faire une boucle sur ListeProduits et d'additionner le prix de chaque produit :

Code : PHP

```
<?php
// Exemple :
class Commande
{
    public function getPrixTotal()
    {
        $prix = 0;
        foreach($this->getListeProduits() as $produit)
        {
            $prix += $produit->getPrix();
        }
        return $prix;
    }
}
```

N'hésitez donc pas à créer des méthodes getQuelquechose() qui contiennent de la logique métier. L'avantage de mettre la logique dans l'entité même est que vous êtes sûrs de réutiliser cette même logique partout dans votre application. Il est bien plus propre et pratique de faire <?php \$commande->getPrixTotal() que d'éparpiller à droite et à gauche différentes manières de calculer ce prix total. Bien sûr, ces méthodes n'ont pas d'équivalent setQuelquechose(), cela n'a pas de sens !

Attributs par défaut

Vous avez aussi parfois besoin de définir une certaine valeur à vos entités lors de leur création. Or nos entités sont de simples objets PHP, et la création d'un objet PHP fait appel... au constructeur. Pour notre entité Article, on pourrait définir le

constructeur suivant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Article
{
    // La définition des attributs...

    public function __construct()
    {
        $this->date = new \Datetime(); // Par défaut, la date de
        l'article est la date d'aujourd'hui
    }

    // Les getters/setters...
}
```

Conclusion

N'oubliez pas : une entité est un objet PHP qui correspond à un besoin dans votre application.

N'essayez donc pas de raisonner en termes de tables, base de données, etc. Vous travaillez maintenant avec des objets PHP, qui contiennent une part de logique métier, et qui peuvent se manipuler facilement. C'est vraiment important que vous fassiez l'effort dès maintenant de prendre l'habitude de manipuler des objets, et non des tables.

Tout sur le mapping !

Vous avez rapidement vu comment *mapper* vos objets avec les annotations. Mais ces annotations permettent d'inscrire pas mal d'autres informations. Il faut juste en connaître la syntaxe, c'est l'objectif de cette section.

Tout ce qui va être décrit ici se trouve bien entendu dans [la documentation officielle sur le mapping](#), que vous pouvez garder à portée de main.

L'annotation Entity

L'annotation `Entity` s'applique sur une classe, il faut donc la placer avant la définition de la classe en PHP. Elle définit un objet comme étant une entité, et donc persisté par Doctrine. Cette annotation s'écrit comme suit :

Code : Autre

```
@ORM\Entity
```

Il existe un seul paramètre facultatif pour cette annotation, `repositoryClass`. Il permet de préciser le namespace complet du repository qui gère cette entité. Nous donnerons le même nom à nos repositories qu'à nos entités, en les suffixant simplement de « `Repository` ». Pour notre entité `Article`, cela donne :

Code : Autre

```
@ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
```



Un *repository* sert à récupérer vos entités depuis la base de données, on en reparle dans un chapitre dédié plus loin dans le cours.

L'annotation Table

L'annotation `Table` s'applique sur une classe également. C'est une annotation facultative, une entité se définit juste par son annotation `Entity`. Cependant, l'annotation `Table` permet de personnaliser le nom de la table qui sera créée dans la base de données. Par exemple, on pourrait préfixer notre table `article` par « `sdz` » :

Code : Autre

```
@ORM\Table(name="sdz_article")
```

Elle se positionne juste avant la définition de la classe.



Par défaut, si vous ne précisez pas cette annotation, le nom de la table créée par Doctrine2 est le même que celui de l'entité. Dans notre cas, cela aurait été « `Article` », avec la majuscule donc, attention car la convention de nommage des tables d'une base de données est de ne pas employer de majuscule. Pensez aussi que si vous êtes sous Windows cela n'a pas d'importance, mais quand vous déployerez votre site sur un serveur sous Linux, bonjour les erreurs de casse !

L'annotation Column

L'annotation `Column` s'applique sur un attribut de classe, elle se positionne donc juste avant la définition PHP de l'attribut concerné. Cette annotation permet de définir les caractéristiques de la colonne concernée. Elle s'écrit comme suit :

Code : Autre

```
@ORM\Column
```

L'annotation `Column` comprend quelques paramètres, dont le plus important est le type de la colonne.

Les types de colonnes

Les types de colonnes que vous pouvez définir en annotation sont des types Doctrine, *et uniquement Doctrine*. Ne les confondez pas avec leurs homologues SQL ou PHP, ce sont des types à Doctrine seul. Ils font la transition des types SQL aux types PHP.

Voici dans le tableau suivant la liste exhaustive des types Doctrine2 disponibles.

Type Doctrine	Type SQL	Type PHP	Utilisation
string	VARCHAR	string	Toutes les chaînes de caractères jusqu'à 255 caractères.
integer	INT	integer	Tous les nombres jusqu'à 2 147 483 647.
smallint	SMALLINT	integer	Tous les nombres jusqu'à 32 767.
bigint	BIGINT	string	Tous les nombres jusqu'à 9 223 372 036 854 775 807. Attention, PHP reçoit une chaîne de caractères, car il ne supporte pas un si grand nombre (suivant que vous êtes en 32 ou en 64 bits).

boolean	BOOLEAN	boolean	Les valeurs booléennes true et false.
decimal	DECIMAL	double	Les nombres à virgule.
date ou datetime	DATETIME	objet DateTime	Toutes les dates et heures.
time	TIME	objet DateTime-	Toutes les heures.
text	CLOB	string	Les chaînes de caractères de plus de 255 caractères.
object	CLOB	Type de l'objet stocké	Stocke un objet PHP en utilisant serialize/unserialize.
array	CLOB	array	Stocke un tableau PHP en utilisant serialize/unserialize.
float	FLOAT	double	Tous les nombres à virgule. Attention, fonctionne uniquement sur les serveurs dont la locale utilise un point comme séparateur.



Les types Doctrine sont sensibles à la casse. Ainsi, le type « String » n'existe pas, il s'agit du type « string ». Facile à retenir : tout est en minuscule !

Le type de colonne se définit en tant que paramètre de l'annotation `Column`, comme suit :

Code : Autre

```
@ORM\Column(type="string")
```

Les paramètres de l'annotation `Column`

Il existe 7 paramètres, tous facultatifs, que l'on peut passer à l'annotation `Column` afin de personnaliser le comportement. Voici la liste exhaustive dans le tableau suivant.

Paramètre	Valeur par défaut	Utilisation
type	string	Définit le type de colonne comme nous venons de le voir.
name	Nom de l'attribut	Définit le nom de la colonne dans la table. Par défaut, le nom de la colonne est le nom de l'attribut de l'objet, ce qui convient parfaitement. Mais vous pouvez changer le nom de la colonne, par exemple si vous préférez « isExpired » en attribut, mais « is_expired » dans la table.
length	255	Définit la longueur de la colonne. Applicable uniquement sur un type de colonne <code>string</code> .
unique	false	Définit la colonne comme unique. Par exemple sur une colonne e-mail pour vos membres.
nullable	false	Permet à la colonne de contenir des NULL.
precision	0	Définit la précision d'un nombre à virgule, c'est-à-dire le nombre de chiffres en tout. Applicable uniquement sur un type de colonne <code>decimal</code> .
scale	0	Définit le <i>scale</i> d'un nombre à virgule, c'est-à-dire le nombre de chiffres après la virgule. Applicable uniquement sur un type de colonne <code>decimal</code> .

Pour définir plusieurs options en même temps, il faut simplement les séparer avec une virgule. Par exemple, pour une colonne « e-mail » en `string` 255 et unique, il faudra faire :

Code : Autre

```
@ORM\Column(type="string", length=255, unique=true)
```

Pour conclure

Vous savez maintenant tout ce qu'il faut savoir sur la couche Modèle sous Symfony2 en utilisant les entités de l'ORM Doctrine2.

Je vous redonne l'adresse de la documentation Doctrine2, que vous serez amenés à utiliser maintes fois dans vos développements : <http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/index.html>. J'insiste : enregistrez-la dans vos favoris, car Doctrine est une bibliothèque très large, et bien que je vous donne un maximum d'informations dans cette partie du tutoriel, je ne pourrai pas tout couvrir.

Attention, Doctrine étant une bibliothèque totalement indépendante de Symfony2, sa documentation fait référence à ce type d'annotation : `/** @Entity */`.

Il faut impérativement l'adapter à votre projet Symfony2, en préfixant toutes les annotations par « `ORM\` » comme nous l'avons vu dans ce chapitre : `/** @ORM\Entity */`.

Car dans nos entités, c'est le namespace `ORM` que nous chargeons. Ainsi, l'annotation `@Entity` n'existe pas pour nous, c'est `@ORM` qui existe (et tous ses enfants : `@ORM\Entity`, `@ORM\Table`, etc.).

Dans le prochain chapitre, nous apprendrons à manipuler les entités que nous savons maintenant construire.

En résumé

- Le rôle d'un ORM est de se charger de la persistance de vos données : vous manipulez des objets, et lui s'occupe de les enregistrer en base de données.
- L'ORM par défaut livré avec Symfony2 est Doctrine2.
- L'utilisation d'un ORM implique un changement de raisonnement : on utilise des objets, et on raisonne en POO. C'est au développeur de s'adapter à Doctrine2, et non l'inverse !
- Une entité est, du point de vue PHP, un simple objet. Du point de vue de Doctrine, c'est un objet complété avec des informations de *mapping* qui lui permettent d'enregistrer correctement l'objet en base de données.
- Une entité est, du point de vue de votre code, un objet PHP qui correspond à un besoin, et indépendant du reste de votre application.



Manipuler ses entités avec Doctrine2

Le chapitre précédent nous a permis d'apprendre à construire des entités. Mais une fois les entités établies, il faut les manipuler !

L'objectif de ce chapitre est donc de voir comment on manipule des entités à l'aide de Doctrine2. Dans un premier temps, nous verrons comment synchroniser les entités avec leur représentation en tables que Doctrine utilise, car en effet, à chaque changement dans une entité, il faut bien que Doctrine mette également à jour la base de données ! Ensuite, nous verrons comment bien manipuler les entités : modification, suppression, etc. Enfin, je vous donnerai un aperçu de la façon de récupérer ses entités depuis la base de données, avant d'aborder cette notion dans un prochain chapitre dédié.

Matérialiser les tables en base de données

Avant de pouvoir utiliser notre entité comme il se doit, on doit d'abord créer la table correspondante dans la base de données !

Créer la table correspondante dans la base de données

Alors, j'espère que vous avez installé et configuré phpMyAdmin, on va faire de la requête SQL !

...

Ceux qui m'ont cru, relisez le chapitre précédent. 🍷 Les autres, venez, on est bien trop fainéants pour ouvrir phpMyAdmin !

Avant toute chose, vérifiez que vous avez bien configuré l'accès à votre base de données dans Symfony2. Si ce n'est pas le cas, il suffit d'ouvrir le fichier `app/config/parameters.yml` et de mettre les bonnes valeurs aux lignes commençant par `database_`: serveur, nom de la base, nom d'utilisateur et mot de passe. Vous avez l'habitude de ces paramètres, voici les miens, mais adaptez-les à votre cas :

Code : YAML

```
# app/config/parameters.yml

parameters:
    database_driver:    pdo_mysql
    database_host:      localhost
    database_port:      ~
    database_name:      symfony
    database_user:      root
    database_password: ~
```

Ensuite, direction la console. Cette fois-ci, on ne va pas utiliser une commande du `generator`, mais une commande de **Doctrine**, car on ne veut pas générer du code mais une table dans la base de données.

D'abord, si vous ne l'avez pas déjà fait, il faut créer la base de données. Pour cela, exécutez la commande (vous n'avez à le faire qu'une seule fois évidemment) :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:database:create
Created database for connection named `symfony`  
C:\wamp\www\Symfony>_
```

Ensuite, il faut générer les tables à l'intérieur de cette base de données. Exécutez donc la commande suivante :

Code : Console

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

Cette dernière commande est vraiment performante. Elle va comparer l'état actuel de la base de données avec ce qu'elle devrait

être en tenant compte de toutes nos entités. Puis elle affiche les requêtes SQL à exécuter pour passer de l'état actuel au nouvel état.

En l'occurrence, nous avons seulement créé une entité, donc la différence entre l'état actuel (base de données vide) et le nouvel état (base de données avec une table Article) n'est que d'une seule requête SQL : la requête de création de la table. Doctrine vous affiche donc cette requête :

Code : SQL

```
CREATE TABLE Article (id INT AUTO_INCREMENT NOT NULL,
date DATETIME NOT NULL,
titre VARCHAR(255) NOT NULL,
auteur VARCHAR(255) NOT NULL,
contenu LONGTEXT NOT NULL,
PRIMARY KEY(id)) ENGINE = InnoDB;
```

Pour l'instant, rien n'a été fait en base de données, Doctrine nous a seulement affiché la ou les requêtes qu'il s'apprête à exécuter. Pensez à toujours valider rapidement ces requêtes, pour être sûrs de ne pas avoir fait d'erreur dans le *mapping* des entités. Mais maintenant, il est temps de passer aux choses sérieuses, et d'exécuter concrètement cette requête ! Lancez la commande suivante :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:schema:update --force
Updating database schema...
Database schema updated successfully! "1" queries were executed
C:\wamp\www\Symfony>_
```

Si tout se passe bien, vous avez le droit au `Database schema updated successfully!`. Génial, mais bon, vérifions-le quand même. Cette fois-ci, ouvrez phpMyAdmin (vraiment, ce n'est pas un piège), allez dans votre base de données et voyez le résultat : la table Article a bien été créée avec les bonnes colonnes, l'id en auto-incrémentation, etc. C'est super !

Modifier une entité

Pour modifier une entité, il suffit de lui créer un attribut et de lui attacher l'annotation correspondante. Faisons-le dès maintenant en ajoutant un attribut `$publication`, un booléen qui indique si l'article est publié (`true` pour l'afficher sur la page d'accueil, `false` sinon), ce n'est qu'un exemple bien entendu. Rajoutez donc ces lignes dans votre entité :

Code : PHP

```
<?php
// sdz/Sdz/BlogBundle/Entity/Article.php

class Article
{
    // ...

    /**
     * @ORM\Column(name="publication", type="boolean")
     */
    private $publication;

    // Et modifions le constructeur pour mettre cet attribut
    // publication à true par défaut
    public function __construct()
    {
        $this->date = new \Datetime();
        $this->publication = true;
    }
}
```

```
    }  
    // ...  
}
```

Ensuite, soit vous écrivez vous-mêmes le getter `getPublication` et le setter `setPublication`, soit vous faites comme moi et vous utilisez le générateur !

Après la commande `doctrine:generate:entity` pour générer une entité entière, vous avez la commande `doctrine:generate:entities`. C'est une commande qui génère les entités en fonction du *mapping* que Doctrine connaît. Lorsque vous faites votre *mapping* en YAML, il peut générer toute votre entité. Dans notre cas, nous faisons notre *mapping* en annotation, alors nous avons déjà défini l'attribut. La commande va donc générer ce qu'il manque : le getter et le setter !

Allons-y :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:generate:entities SdzBlogBundle:Article  
Generating entity "Sdz\BlogBundle\Entity\Article"  
  > backing up Article.php to Article.php~  
  > generating Sdz\BlogBundle\Entity\Article
```

Allez vérifier votre entité, tout en bas de la classe, le générateur a rajouté les méthodes `getPublication()` et `setPublication()`.

 Vous pouvez voir également qu'il a sauvegardé l'ancienne version de votre entité dans un fichier nommé `Article.php~` : vérifiez toujours son travail, et si celui-ci ne vous convient pas, vous avez votre sauvegarde. 

Maintenant, il ne reste plus qu'à enregistrer ce schéma en base de données. Exécutez donc :

Code : Console

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

... pour vérifier que la requête est bien :

Code : SQL

```
ALTER TABLE article ADD publication TINYINT(1) NOT NULL
```

C'est le cas, cet outil de Doctrine est vraiment pratique ! Puis exécutez la commande pour modifier effectivement la table correspondante :

Code : Console

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

Et voilà ! Votre entité a un nouvel attribut qui sera persisté en base de données lorsque vous l'utiliserez.

À retenir

À chaque modification du *mapping* des entités, ou lors de l'ajout/suppression d'une entité, il faudra répéter ces commandes `doctrine:schema:update --dump-sql` puis `--force` pour mettre à jour la base de données.

Enregistrer ses entités avec l'EntityManager

Maintenant, apprenons à manipuler nos entités. On va apprendre à le faire en deux parties : d'abord l'enregistrement en base de données, ensuite la récupération depuis la base de données. Mais d'abord, étudions un petit peu le service Doctrine.

Les services Doctrine2

Rappelez-vous, un service est une classe qui remplit une fonction bien précise, accessible partout dans notre code. Dans ce paragraphe, concentrons-nous sur ce qui nous intéresse : accéder aux fonctionnalités Doctrine2 via leurs services.

Le service Doctrine

Le service Doctrine est celui qui va nous permettre de gérer la persistance de nos objets. Ce service est accessible depuis le contrôleur comme n'importe quel service :

Code : PHP

```
<?php  
$doctrine = $this->get('doctrine');
```

Mais, afin de profiter de l'autocomplétion de votre IDE, la classe Controller de Symfony2 intègre un raccourci. Il fait exactement la même chose, mais est plus joli et permet l'autocomplétion :

Code : PHP

```
<?php  
$doctrine = $this->getDoctrine();
```

C'est donc ce service Doctrine qui va nous permettre de gérer la base de données. Il permet de gérer deux choses :

- Les différentes connexions à des bases de données. C'est la partie DBAL de Doctrine2. En effet, vous pouvez tout à fait utiliser plusieurs connexions à plusieurs bases de données différentes. Cela n'arrive que dans des cas particuliers, mais c'est toujours bon à savoir que Doctrine le gère bien. Le service Doctrine dispose donc, entre autres, de la méthode `$doctrine->getConnection ($name)` qui permet de récupérer une connexion à partir de son nom. Cette partie DBAL permet à Doctrine2 de fonctionner sur plusieurs types de SGBDR, tels que MySQL, PostgreSQL, etc.
- Les différents gestionnaires d'entités, ou EntityManager. C'est la partie ORM de Doctrine2. Encore une fois, c'est logique, vous pouvez bien sûr utiliser plusieurs gestionnaires d'entités, ne serait-ce qu'un par connexion ! Le service dispose donc, entre autres, de la méthode dont nous nous servirons beaucoup : `$doctrine->getManager ($name)` qui permet de récupérer un ORM à partir de son nom.



Dans la suite du tutoriel, je considère que vous n'avez qu'un seul EntityManager, ce qui est le cas par défaut. La méthode `getManager ()` permet de récupérer l'EntityManager par défaut en omettant l'argument `$name`. J'utiliserai donc toujours `$doctrine->getManager ()` sans argument, mais pensez à adapter si ce n'est pas votre cas !



Si vous souhaitez utiliser plusieurs EntityManager, vous pouvez vous référer à la documentation officielle qui l'explique.

Le service EntityManager

On vient de le voir, le service qui va nous intéresser vraiment n'est pas doctrine, mais l'EntityManager de Doctrine. Vous savez déjà le récupérer depuis le contrôleur via :

Code : PHP

```
<?php
$em = $this->getDoctrine()->getManager();
```



Pour ceux qui viennent de Symfony2.0, on utilisait avant `->getEntityManager()`, mais depuis la version 2.1 son accès a été simplifié au profit de `->getManager()`.

Mais sachez que, comme tout service qui se respecte, vous pouvez y accéder directement via :

Code : PHP

```
<?php
$em = $this->get('doctrine.orm.entity_manager');
```

Mais attention, la première méthode vous assure l'autocomplétion alors que la deuxième non. 😊

C'est avec l'EntityManager que l'on va passer le plus clair de notre temps. C'est lui qui permet de dire à Doctrine « Persiste cet objet », c'est lui qui va exécuter les requêtes SQL (que l'on ne verra jamais), bref, c'est lui qui fera tout.

La seule chose qu'il ne sait pas faire facilement, c'est récupérer les entités depuis la base de données. Pour faciliter l'accès aux objets, on va utiliser des *Repository*.

Les repositories

Les *repositories* sont des objets, qui utilisent un EntityManager en les coulisses, mais qui sont bien plus faciles et pratiques à utiliser de notre point de vue. Je parle des repositories au pluriel car il en existe **un par entité**. Quand on parle d'un repository en particulier, il faut donc toujours préciser le repository de quelle entité, afin de bien savoir de quoi on parle.

On accède à ces repositories de la manière suivante :

Code : PHP

```
<?php
$em = $this->getDoctrine()->getManager();
getRepository_article = $em->getRepository('SdzBlogBundle:Article');
```

L'argument de la méthode `getRepository` est l'entité pour laquelle récupérer le repository. Il y a deux manières de spécifier l'entité voulue :

- Soit en utilisant le namespace complet de l'entité. Pour notre exemple, cela donnerait :
`'Sdz\BlogBundle\Entity\Article'.`
- Soit en utilisant le raccourci `Nom_du_bundle:Nom_de_l'` entité. Pour notre exemple, c'est donc
`'SdzBlogBundle:Article'.` C'est un raccourci qui fonctionne partout dans Doctrine.



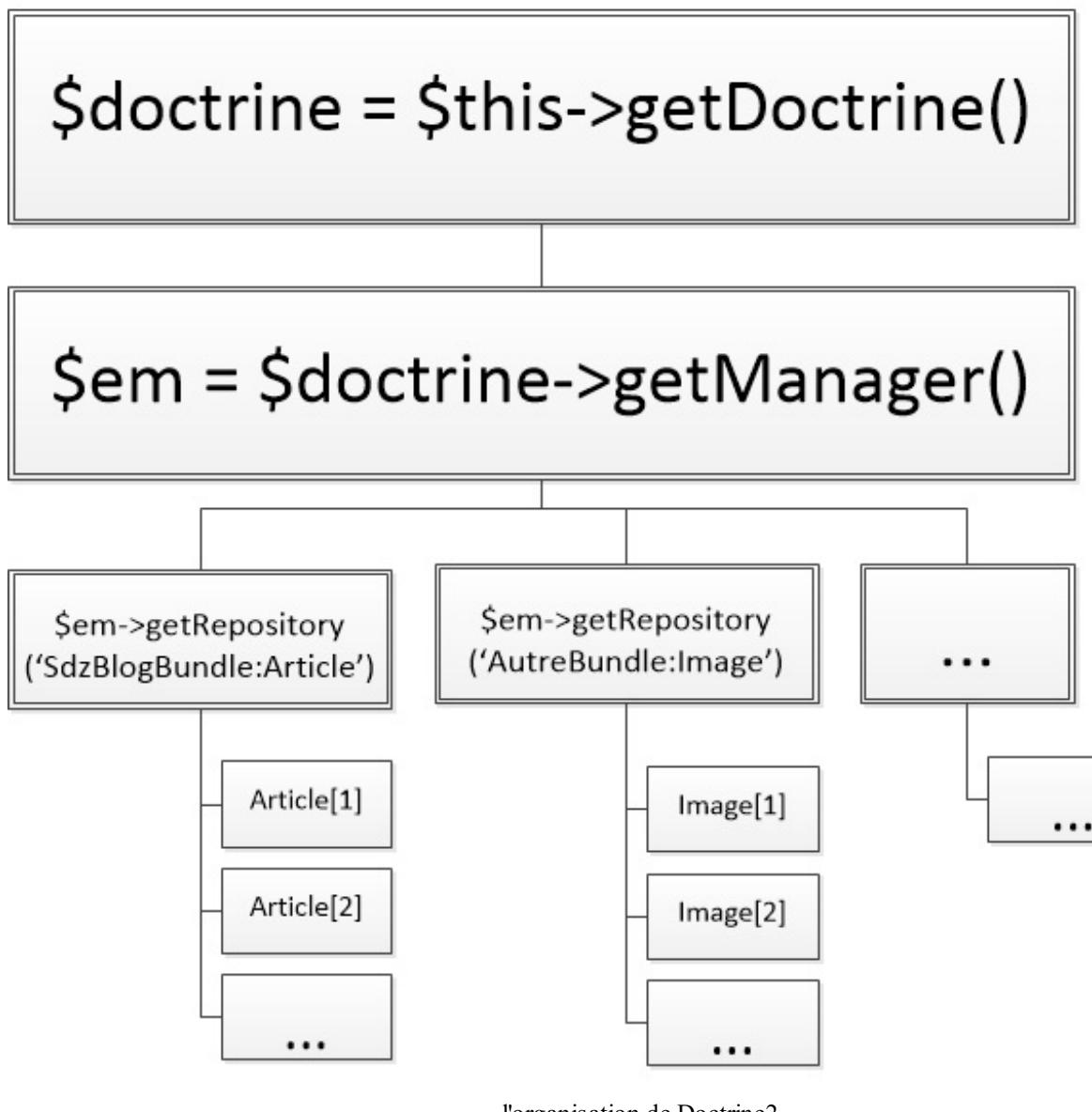
Attention, ce raccourci ne fonctionne que si vous avez mis vos entités dans le namespace Entity dans votre bundle.



Ce sont donc ces repositories qui nous permettront de récupérer nos entités. Ainsi, pour charger deux entités différentes, il faut d'abord récupérer leur repository respectif. Un simple pli à prendre, mais très logique.

Conclusion

Vous savez maintenant accéder aux principaux acteurs que nous allons utiliser pour manipuler nos entités. Ils reviendront très souvent, sachez les récupérer par cœur, cela vous facilitera la vie. Afin de bien les visualiser, je vous propose à la figure suivante un petit schéma à avoir en tête.



Enregistrer ses entités en base de données

Rappelez-vous, on a déjà vu comment créer une entité. Maintenant que l'on a cette magnifique entité entre les mains, il faut la donner à Doctrine pour qu'il l'enregistre en base de données. L'enregistrement effectif en base de données se fait en deux étapes très simples depuis un contrôleur. Modifiez la méthode `ajouterAction ()` de notre contrôleur pour faire les tests :

Code : PHP

```
<?php
```

```
// src/Sdz/BlogBundle/Controller/BlogController.php

// Attention à bien ajouter ce use en début de contrôleur
use Sdz\BlogBundle\Entity\Article;

// ...

public function ajouterAction()
{
    // Création de l'entité
    $article = new Article();
    $article->setTitre('Mon dernier weekend');
    $article->setAuteur('Bibi');
    $article->setContenu("C'était vraiment super et on s'est bien
    amusé.");
    // On peut ne pas définir ni la date ni la publication,
    // car ces attributs sont définis automatiquement dans le
    constructeur

    // On récupère l'EntityManager
    $em = $this->getDoctrine()->getManager();

    // Étape 1 : On « persiste » l'entité
    $em->persist($article);

    // Étape 2 : On « flush » tout ce qui a été persisté avant
    $em->flush();

    // Reste de la méthode qu'on avait déjà écrit
    if ($this->getRequest()->getMethod() == 'POST') {
        $this->get('session')->setFlashBag()->add('info', 'Article
        bien enregistré');
        return $this->redirect( $this->generateUrl('sdzblog_voir',
        array('id' => $article->getId())));
    }

    return $this->render('SdzBlogBundle:Blog:ajouter.html.twig');
}
```

Reprendons ce code :

- La ligne 12 permet de créer l'entité, et les lignes 13 à 15 de renseigner ses attributs ;
- La ligne 20 permet de récupérer l'EntityManager, on en a déjà parlé, je ne reviens pas dessus ;
- L'étape 1 dit à Doctrine de « persister » l'entité. Cela veut dire qu'à partir de maintenant cette entité (qui n'est qu'un simple objet !) est gérée par Doctrine. Cela n'exécute pas encore de requête SQL, ni rien d'autre.
- L'étape 2 dit à Doctrine d'exécuter effectivement les requêtes nécessaires pour sauvegarder les entités qu'on lui a dit de persister ;
- Ligne 31, notre Article étant maintenant enregistré en base de données grâce au `flush()`, Doctrine2 lui a attribué un id !

Allez sur la page [/blog/ajouter](#), et voilà, vous venez d'ajouter un article dans la base de données !

Si la requête SQL effectuée vous intéresse, je vous invite à cliquer sur l'icône tout à droite dans la barre d'outil Symfony2 en bas de la page, comme le montre la figure suivante.



Vous arrivez alors dans la partie Doctrine du *profiler* de Symfony2, et vous pouvez voir les différentes requêtes SQL exécutées

par Doctrine. C'est très utile pour vérifier la valeur des paramètres, la structure des requêtes, etc. N'hésitez pas à y faire des tours !

Queries

Connection default

```
+ SET NAMES UTF8
```

Parameters: {}

Time: 0.14 ms

```
INSERT INTO Article (date, titre, contenu) VALUES (?, ?, ?)
```

Parameters: { 1: Object(DateTime), 2: "Mon dernier weekend", 3: "C'était vraiment super et on s'est bien amusé." }

Time: 31.96 ms

Database Connections

On voit la requête

Key	Value
default	doctrine.dbal.default_connection

Entity Managers

Key	Value
default	doctrine.orm.default_entity_manager

effectuée

Alors, vous me direz qu'ici on n'a persisté qu'une seule entité, c'est vrai. Mais on peut tout à fait faire plusieurs persists sur différentes entités avant d'exécuter un seul flush. Le flush permet d'exécuter les requêtes les plus optimisées pour enregistrer tous nos persists.

Doctrine utilise les transactions

Pourquoi deux méthodes `$em->persist()` et `$em->flush()` ? Car cela permet entre autres de profiter des **transactions**. Imaginons que vous ayez plusieurs entités à persister en même temps. Par exemple, lorsque l'on crée un sujet sur un forum, il faut enregistrer l'entité `Sujet`, mais aussi l'entité `Message`, les deux en même temps. Sans transaction, vous feriez d'abord la première requête, puis la deuxième. Logique au final. Mais imaginez que vous ayez enregistré votre `Sujet`, et que l'enregistrement de votre `Message` échoue : vous avez un sujet sans message ! Cela casse votre base de données, car la relation n'est plus respectée.

Avec une transaction, les deux entités sont enregistrées *en même temps*, ce qui fait que si la deuxième échoue, alors la première est annulée, et vous gardez une base de données propre.

Concrètement, avec notre EntityManager, chaque `$em->persist()` est équivalent à dire : « Garde cette entité en mémoire, tu l'enregistreras au prochain `flush()`. » Et un `$em->flush()` est équivalent à : « Ouvre une transaction et enregistre toutes les entités qui t'ont été données depuis le dernier `flush()`. »

Doctrine simplifie la vie

Vous devez savoir une chose également : la méthode `$em->persist()` traite indifféremment les nouvelles entités de celles déjà en base de données. Vous pouvez donc lui passer une entité fraîchement créée comme dans notre exemple précédent, mais également une entité que vous auriez récupérée grâce à l'EntityRepository et que vous auriez modifiée (ou non, d'ailleurs). L'EntityManager s'occupe de tout, je vous disais !

Concrètement, cela veut dire que vous n'avez plus à vous soucier de faire des **INSERT INTO** dans le cas d'une création d'entité, et des **UPDATE** dans le cas d'entités déjà existantes. Exemple :

Code : PHP

```
<?php
// Depuis un contrôleur

$em = $this->getDoctrine()->getManager();

// On crée un nouvel article
$article1 = new Article;
$article1->setTitre('Mon dernier weekend');
$article1->setContenu("C'était vraiment super et on s'est bien
amusé.");
// Et on le persiste
$em->persist($article1);

// On récupère l'article d'id 5. On n'a pas encore vu cette méthode
// find(), mais elle est simple à comprendre
// Pas de panique, on la voit en détail dans un prochain chapitre
// dédié aux repositories
$article2 = $em->getRepository('SdzBlogBundle:Article')->find(5);

// On modifie cet article, en changeant la date à la date
// d'aujourd'hui
$article2->setDate(new \Datetime());

// Ici, pas besoin de faire un persist() sur $article2. En effet,
// comme on a récupéré cet article via Doctrine,
// il sait déjà qu'il doit gérer cette entité. Rappelez-vous, un
// persist ne sert qu'à donner la responsabilité de l'objet à
// Doctrine.

// Enfin, on applique les changements à la base de données
$em->flush();
```

Le `flush()` va donc exécuter un **INSERT INTO** et un **UPDATE** tout seul. De notre côté, on a traité `$article1` exactement comme `$article2`, ce qui nous simplifie bien la vie. Comment sait-il si l'entité existe déjà ou non ? Grâce à la clé primaire de votre entité (dans notre cas, l'`id`). Si l'`id` est nul, c'est une nouvelle entité, tout simplement. 😊

Retenez bien également le fait qu'il est inutile de faire un `persist($entité)` lorsque `$entité` a été récupérée grâce à Doctrine. En effet, rappelez-vous qu'un `persist` ne fait rien d'autre que de donner la responsabilité d'un objet à Doctrine. Dans le cas de la variable `$article1` de l'exemple précédent, Doctrine ne peut pas deviner qu'il doit s'occuper de cet objet si on ne le lui dit pas ! D'où le `persist()`. Mais à l'inverse, comme c'est Doctrine qui nous a donné l'objet `$article2`, il est grand et prend tout seul la responsabilité de cet objet, inutile de le lui répéter.

Sachez également que Doctrine est assez intelligent pour savoir si une entité a été modifiée ou non. Ainsi, si dans notre exemple on ne modifiait pas `$article2`, Doctrine ne ferait pas de requête **UPDATE** inutile.

Les autres méthodes utiles de l'EntityManager

En plus des deux méthodes les plus importantes, `persist()` et `flush()`, l'EntityManager dispose de quelques méthodes intéressantes. Je ne vais vous présenter ici que les plus utilisées, mais elles sont bien sûr toutes documentées dans la [documentation officielle](#), que je vous invite fortement à aller voir.

`clear($nomEntité)` annule tous les `persist()` effectués. Si le nom d'une entité est précisé (son namespace complet ou son raccourci), seuls les `persist()` sur des entités de ce type seront annulés. Voici un exemple :

Code : PHP

```
<?php
$em->persist($article);
$em->persist($commentaire);
$em->clear();
$em->flush(); // N'exécutera rien, car les deux persists sont
annulés par le clear
```

detach(\$entité) annule le `persist()` effectué sur l'entité en argument. Au prochain `flush()`, aucun changement ne sera donc appliqué à l'entité. Voici un exemple :

Code : PHP

```
<?php
$em->persist($article);
$em->persist($commentaire);
$em->detach($article);
$em->flush(); // Enregistre $commentaire mais pas $article
```

contains(\$entité) retourne `true` si l'entité donnée en argument est gérée par l'EntityManager (s'il y a eu un `persist()` sur l'entité donc). Voici un exemple :

Code : PHP

```
<?php
$em->persist($article);
var_dump($em->contains($article)); // Affiche true
var_dump($em->contains($commentaire)); // Affiche false
```

refresh(\$entité) met à jour l'entité donnée en argument dans l'état où elle est en base de données. Cela écrase et donc annule tous les changements qu'il a pu y avoir sur l'entité concernée. Voici un exemple :

Code : PHP

```
<?php
$article->setTitre('Un nouveau titre');
$em->refresh($article);
var_dump($article->getTitre()); // Affiche « Un ancien titre »
```

remove(\$entité) supprime l'entité donnée en argument de la base de données. Effectif au prochain `flush()`. Voici un exemple :

Code : PHP

```
<?php
$em->remove($article);
$em->flush(); // Exécute un DELETE sur $article
```

Récupérer ses entités avec un EntityRepository

Un prochain chapitre entier est consacré aux repositories, juste après dans cette partie sur Doctrine. Les repositories ne sont qu'un outil pour récupérer vos entités très facilement, nous apprendrons à les maîtriser entièrement. Mais en avant première, sachez au moins récupérer une unique entité en fonction de son id.

Il faut d'abord pour cela récupérer le repository de l'entité que vous voulez. On l'a vu précédemment, voici un rappel :

Code : PHP

```
<?php
// Depuis un contrôleur

$repository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');
```

Puis depuis ce repository, il faut utiliser la méthode `find($id)` qui permet de retourner l'entité correspondant à l'id `$id`. Je vous invite à essayer ce code directement dans la méthode `voirAction()` de notre contrôleur Blog, là où on avait défini en dur un tableau `$article`. On pourra ainsi voir l'effet immédiatement :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...

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

    // On récupère l'entité correspondant à l'id $id
    $article = $repository->find($id);

    // $article est donc une instance de
    // Sdz\BlogBundle\Entity\Article

    // Ou null si aucun article n'a été trouvé avec l'id $id
    if($article === null)
    {
        throw $this->createNotFoundException('Article[id='.$id.']'
            . 'inexistant.');
    }

    return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
        'article' => $article
    ));
}
```



Allez voir le résultat sur la page [/blog/article/1](#) ! Vous pouvez changer l'id de l'article à récupérer dans l'URL, en fonction des articles que vous avez ajoutés plus haut depuis la méthode `ajouterAction()`.

Sachez aussi qu'il existe une autre syntaxe pour faire la même chose directement depuis l'EntityManager, je vous la présente afin que vous ne soyez pas surpris si vous la croisez :

Code : PHP

```
<?php
// Depuis un contrôleur
```

```
$article = $this->getDoctrine()  
          ->getManager()  
          ->find('SdzBlogBundle:Article', $id); // 1er  
argument : le nom de l'entité  
           // 2e argument  
: l'id de l'instance à récupérer
```

Je n'en dis pas plus pour le moment, patientez jusqu'au chapitre consacré aux repositories !

En résumé

- Il faut exécuter la commande `doctrine:schema:update` pour mettre à jour la base de données et la faire correspondre à l'état actuel de vos entités ;
- Avec Symfony2, on récupère l'EntityManager de Doctrine2 via le service `doctrine.orm.entity_manager` ou, plus simplement depuis un contrôleur, via `$this->getDoctrine()->getManager()` ;
- L'EntityManager sert à **manipuler** les entités, tandis que les repositories servent à **récupérer** les entités.

Les relations entre entités avec Doctrine2

Maintenant que vous savez créer et manipuler une entité simple, on va monter en puissance.

L'objectif de ce chapitre est de construire un ensemble d'entités en relation les unes avec les autres. Ces relations permettent de disposer d'un ensemble cohérent, qui se manipule simplement et en toute sécurité pour votre base de données.

Présentation

Présentation

Vous savez déjà stocker vos entités indépendamment les unes des autres, c'est très bien. Simplement, on est rapidement limités ! L'objectif de ce chapitre est de vous apprendre à établir des relations entre les entités.

Rappelez-vous, au début de la partie sur Doctrine2, je vous avais promis des choses comme `<?php $article->getCommentaires()`. Eh bien, c'est cela que nous allons faire ici !

Les différents types de relations

Il y a plusieurs façons de lier des entités entre elles. En effet, il n'est pas pareil de lier une multitude de commentaires à un seul article et de lier un membre à un seul groupe. Il existe donc plusieurs types de relations, pour répondre à plusieurs besoins concrets. Ce sont les relations OneToOne, OneToMany et ManyToMany. On les étudie juste après ces quelques notions de base à avoir.

Notions techniques d'ORM à savoir

Avant de voir en détail les relations, il faut comprendre comment elles fonctionnent. N'ayez pas peur, il y a juste deux notions à savoir avant d'attaquer.

Notion de propriétaire et d'inverse

La notion de propriétaire et d'inverse est abstraite mais importante à comprendre. *Dans une relation entre deux entités, il y a toujours une entité dite **propriétaire**, et une dite **inverse**.* Pour comprendre cette notion, il faut revenir à la vieille époque, lorsque l'on faisait nos bases de données à la main. *L'entité propriétaire est celle qui contient la référence à l'autre entité.* Attention, cette notion — à avoir en tête lors de la création des entités — n'est pas liée à votre logique métier, elle est purement technique.

Prenons un exemple simple, toujours les commentaires d'un article de blog. Vous disposez de la table `commentaire` et de la table `article`. Pour créer une relation entre ces deux tables, vous allez mettre naturellement une colonne `article_id` dans la table `commentaire`. La table `commentaire` est donc propriétaire de la relation, car c'est elle qui contient la colonne de liaison `article_id`. Assez simple au final !



N'allez pas me créer une colonne `article_id` dans la table des commentaires ! C'est une image, de ce que vous faisiez avant. Aujourd'hui on va laisser Doctrine gérer tout ça, et on ne va jamais mettre la main dans PhpMyAdmin. Rappelez-vous : on pense objet dorénavant, et pas base de données.

Notion d'unidirectionnalité et de bidirectionnalité

Cette notion est également simple à comprendre : *une relation peut être à sens unique ou à double sens.* On ne va traiter dans ce chapitre que les relations à sens unique, dites unidirectionnelles. Cela signifie que vous pourrez faire `<?php $entiteProprietaire->getEntiteInverse()` (dans notre exemple `<?php $commentaire->getArticle()`), mais vous ne pourrez pas faire `<?php $entiteInverse->getEntiteProprietaire()` (pour nous, `<?php $article->getCommentaires()`). Attention, cela ne nous empêchera pas de récupérer les commentaires d'un article, on utilisera juste une autre méthode, via l'EntityRepository.

Cette limitation nous permet de simplifier la façon de définir les relations. Pour bien travailler avec, il suffit juste de se rappeler qu'on ne peut pas faire `$entiteInverse->getEntiteProprietaire()`.

Pour des cas spécifiques, ou des préférences dans votre code, cette limitation peut être contournée en utilisant les relations à

double sens, dites bidirectionnelles. Je les expliquerai rapidement à la fin de ce chapitre.

Rien n'est magique

Non, rien n'est magique. Je dois vous avertir qu'un `<?php $article->getCommentaires()` est vraiment sympa, mais qu'il déclenche bien sûr une requête SQL ! Lorsqu'on récupère une entité (notre `$article` par exemple), Doctrine ne récupère pas toutes les entités qui lui sont liées (les commentaires dans l'exemple), et heureusement ! S'il le faisait, cela serait extrêmement lourd. Imaginez qu'on veuille juste récupérer un article pour avoir son titre, et Doctrine nous récupère la liste des 54 commentaires, qui en plus sont liés à leurs 54 auteurs respectifs, etc. !

Doctrine utilise ce qu'on appelle le *Lazy Loading*, « chargement fainéant » en français. C'est-à-dire qu'il ne va charger les entités à l'autre bout de la relation que si vous voulez accéder à ces entités. C'est donc pile au moment où vous faites `<?php $article->getCommentaires()` que Doctrine va charger les commentaires (avec une nouvelle requête SQL donc) puis va vous les transmettre.

Heureusement pour nous, il est possible d'éviter cela ! Parce que cette syntaxe est vraiment pratique, il serait dommage de s'en priver pour cause de requêtes SQL trop nombreuses. Il faudra simplement utiliser nos propres méthodes pour charger les entités, dans lesquelles nous ferons des jointures toutes simples. L'idée est de dire à Doctrine : « Charge l'entité Article, mais également tous ses commentaires ». Avoir nos propres méthodes pour cela permet de ne les exécuter que si nous voulons vraiment avoir les commentaires en plus de l'article. En somme, on se garde le choix de charger ou non la relation.

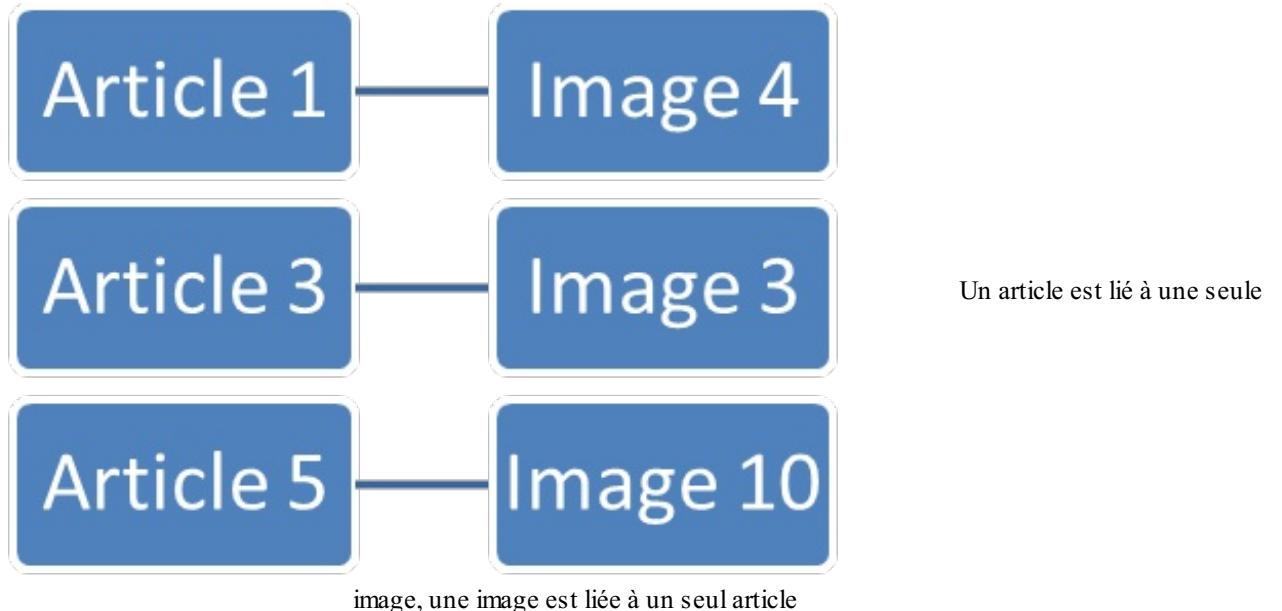
Mais nous verrons tout cela dans le prochain chapitre sur les repositories. Pour l'instant, revenons à nos relations !

Relation One-To-One

Présentation

La relation *One-To-One*, ou **1..1**, est assez classique. Elle correspond, comme son nom l'indique, à une relation unique entre deux objets.

Pour illustrer cette relation dans le cadre du blog, nous allons créer une entité `Image`. Imaginons qu'on offre la possibilité de lier une image à un article, une sorte d'icône pour illustrer un peu l'article. Si à chaque article on ne peut afficher qu'une seule image, et que chaque image ne peut être liée qu'à un seul article, alors on est bien dans le cadre d'une relation One-To-One. La figure suivante schématisé tout cela.



Tout d'abord, histoire qu'on parle bien de la même chose, créez cette entité `Image` avec au moins les attributs `url` et `alt` pour qu'on puisse l'afficher correctement. Voici la mienne :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Image.php
  
```

```

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

< /**
 * Sdz\BlogBundle\Entity\Image
 *
 * @ORM\Table()
 */
@ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ImageRepository")
*/
class Image
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $url
     *
     * @ORM\Column(name="url", type="string", length=255)
     */
    private $url;

    /**
     * @var string $alt
     *
     * @ORM\Column(name="alt", type="string", length=255)
     */
    private $alt;

    // Les getters et setters
}

```

Définition de la relation dans les entités

Annotation

Pour établir une relation One-To-One entre deux entités Article et Image, la syntaxe est la suivante :

Entité propriétaire, Article :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Article.php

< /**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image",
     * cascade={"persist"})
     */
    private $image;

    // ...
}

```

Entité inverse, Image :**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Entity/Image.php

/**
 * @ORM\Entity
 */
class Image
{
    // Nul besoin d'ajouter une propriété ici
    //
}
```

La définition de la relation est plutôt simple, mais détaillons-la bien.

Tout d'abord, j'ai choisi de définir l'entité Article comme entité propriétaire de la relation, car un Article « possède » une Image. On aura donc plus tendance à récupérer l'image à partir de l'article que l'inverse. Cela permet également de rendre indépendante l'entité Image : elle pourra être utilisée par d'autres entités que Article, de façon totalement invisible pour elle.

Ensuite, vous voyez que seule l'entité propriétaire a été modifiée, ici Article. C'est parce qu'on a une relation unidirectionnelle, rappelez-vous, on peut donc faire `$article->getImage()`, mais pas `$image->getArticle()`. Dans une relation unidirectionnelle, l'entité inverse, ici Image, ne sait en fait même pas qu'elle est liée à une autre entité, ce n'est pas son rôle.

Enfin, concernant l'annotation en elle-même :

Code : Autre

```
@ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image", cascade={"persist"})
```

Il y a plusieurs choses à savoir sur cette annotation :

- Elle est incompatible avec l'annotation `@ORM\Column` qu'on a vue dans un chapitre précédent. En effet, l'annotation `Column` définit une valeur (un nombre, une chaîne de caractères, etc.), alors que `OneToOne` définit une relation vers une autre entité. Lorsque vous utilisez l'un, vous ne pouvez pas utiliser l'autre sur le même attribut.
- Elle possède au moins l'option `targetEntity`, qui vaut simplement le namespace complet vers l'entité liée.
- Elle possède d'autres options facultatives, dont l'option `cascade` dont on parle un peu plus loin.



N'oubliez pas de mettre à jour la base de données avec la commande `doctrine:schema:update` !

Rendre une relation facultative

Par défaut, une relation est facultative, c'est-à-dire que vous pouvez avoir un Article qui n'a pas d'Image liée. C'est le comportement que nous voulons pour l'exemple : on se donne le droit d'ajouter un article sans forcément trouver une image d'illustration. Si vous souhaitez forcer la relation, il faut ajouter l'annotation `JoinColumn` et définir son option `nullable` à `false`, comme ceci :

Code : Autre

```
/**
```

```

 * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image")
 * @ORM\JoinColumn(nullable=false)
 */
private $image;

```

Les opérations de cascade

Parlons maintenant de l'option `cascade` que l'on a vu un peu plus haut. Cette option permet de « cascader » les opérations que l'on ferait sur l'entité `Article` à l'entité `Image` liée par la relation.

Pour prendre l'exemple le plus simple, imaginez que vous supprimez une entité `Article` via un `<?php $em->remove($article)`. Si vous ne précisez rien, Doctrine va supprimer l'`Article` mais garder l'entité `Image` liée. Or ce n'est pas forcément ce que vous voulez ! Si vos `images` ne sont liées qu'à des articles, alors la suppression de l'`article` doit entraîner la suppression de l'`image`, sinon vous aurez des `Images` orphelines dans votre base de données. C'est le but de `cascade`. Attention, si vos `images` sont liées à des articles mais aussi à d'autres entités, alors vous ne voulez pas forcément supprimer directement l'`image` d'un article, car elle pourrait être liée à une autre entité.

On peut cascader des opérations de suppression, mais également de persistance. En effet, on a vu qu'il fallait persister une entité avant d'exécuter le `flush()`, afin de dire à Doctrine qu'il doit enregistrer l'entité en base de données. Cependant, dans le cas d'entités liées, si on fait un `$em->persist($article)`, qu'est-ce que Doctrine doit faire pour l'entité `Image` contenue dans l'entité `Article` ? Il ne le sait pas et c'est pourquoi il faut le lui dire : soit en faisant manuellement un `persist()` sur l'`article` et l'`image`, soit en définissant dans l'annotation de la relation qu'un `persist()` sur `Article` doit se « propager » sur l'`Image` liée.

C'est ce que nous avons fait dans l'annotation : on a défini le `cascade` sur l'opération `persist()`, mais pas sur l'opération `remove()` (car on se réserve la possibilité d'utiliser les images pour autre chose que des articles).

Getter et setter

D'abord, n'oubliez pas de définir un `getter` et un `setter` dans l'entité propriétaire, ici `Article`. Vous pouvez utiliser la commande `php app/console doctrine:generate:entities SdzBlogBundle:Article`, ou alors prendre ce code :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image")
     */
    private $image;

    // Vos autres attributs...

    /**
     * @param Sdz\BlogBundle\Entity\Image $image
     */
    public function setImage(\Sdz\BlogBundle\Entity\Image $image = null)
    {
        $this->image = $image;
    }

    /**
     * @return Sdz\BlogBundle\Entity\Image
     */
    public function getImage()

```

```

    {
        return $this->image;
    }

    // Vos autres getters/setters...
}

```

Vous voyez qu'on a forcé le type de l'argument pour le setter `setImage()` : cela permet de déclencher une erreur si vous essayez de passer un autre objet que `Image` à la méthode. Très utile pour éviter de chercher des heures l'origine d'un problème parce que vous avez passé un mauvais argument. Notez également le « = null » qui permet d'accepter les valeurs null : rappelez-vous, la relation est facultative !

Prenez bien conscience d'une chose également : le getter `getImage()` retourne une instance de la classe `Image` directement. Lorsque vous avez un `Article`, disons `$article`, et que vous voulez récupérer l'URL de l'`Image` associée, il faut donc faire :

Code : PHP

```

<?php
$image = $article->getImage();
$url = $image->getUrl();

// Ou bien sûr en plus simple :
$url = $article->getImage()->getUrl();

```

Pour les curieux qui auront été voir ce qui a été fait en base de données : une colonne `image_id` a bien été ajouté à la table `article`. Cependant, ne confondez surtout pas cette colonne `image_id` avec notre attribut `image`, et gardez bien ces deux points en tête :

1/ L'entité `Article` ne contient pas d'attribut `image_id`.

2/ L'attribut `image` ne contient pas l'id de l'`Image` liée, il contient une instance de la classe `Sdz\BlogBundle\Entity\Image` qui, elle, contient un attribut `id`.



N'allez donc jamais m'écrire `$article->getImageId()`, pour récupérer l'id de l'image liée, il faut d'abord récupérer l'`Image` elle-même puis son id, comme ceci : `$article->getImage()->getId()`. Et ne faites surtout pas la confusion : une entité n'est pas une table.

Exemple d'utilisation

Pour utiliser cette relation, c'est très simple. Voici un exemple pour ajouter un nouvel `Article` et son `Image` depuis un contrôleur. Modifions l'action `ajouterAction()`, qui était déjà bien complète :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// N'oubliez pas de rajouter ce use !
use Sdz\BlogBundle\Entity\Image;

// ...

public function ajouterAction()
{
    // Création de l'entité Article
    $article = new Article();
    $article->setTitre('Mon dernier weekend');
    $article->setContenu("C'était vraiment super et on s'est bien
amusé.");
    $article->setAuteur('winzou');
}

```

```

// Création de l'entité Image
$image = new Image();
$image-
>setUrl('http://uploads.siteduzero.com/icones/478001_479000/478657.png');
$image->setAlt('Logo Symfony2');

// On lie l'image à l'article
$article->setImage($image);

// On récupère l'EntityManager
$em = $this->getDoctrine()->getManager();

// Étape 1 : on persiste les entités
$em->persist($article);

// Étape 1 bis : si on n'avait pas défini le cascade={"persist"}, on
devrait persister à la main l'entité $image
// $em->persist($image);

// Étape 2 : on déclenche l'enregistrement
$em->flush();

// ... reste de la méthode
}

```

Pour information, voici comment on pourrait modifier la vue d'un article, pour y intégrer l'image :

Code : HTML & Django

```

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

{% block title %}
    Lecture d'un article - {{ parent() }}
{% endblock %}

{% block sdzblog_body %}

    <h2>
        {% On vérifie qu'une image soit bien associée à l'article %}
        {% if article.image is not null %}
            
        {% endif %}

        {{ article.titre }}
    </h2>
    <i>Par {{ article.auteur }}, le {{ article.date|date('d/m/Y') }}</i>

    <div class="well">
        {{ article.contenu }}
    </div>

    <p>
        <a href="{{ path('sdzblog_accueil') }}" class="btn">
            <i class="icon-chevron-left"></i>
            Retour à la liste
        </a>
        <a href="{{ path('sdzblog_modifier', { 'id': article.id }) }}" class="btn">
            <i class="icon-edit"></i>
            Modifier l'article
        </a>
        <a href="{{ path('sdzblog_supprimer', { 'id': article.id }) }}">

```

```

    class="btn">
        <i class="icon-trash"></i>
        Supprimer l'article
    </a>
</p>

{ % endblock %}

```

Et voici un autre exemple, qui modifierait l'Image d'un Article déjà existant. Ici je vais prendre une méthode arbitraire, mais vous savez tout ce qu'il faut pour l'implémenter réellement :

Code : PHP

```

<?php
// Dans un contrôleur, celui que vous voulez

public function modifierImageAction($id_article)
{
    $em = $this->getDoctrine()->getManager();

    // On récupère l'article
    $article = $em->getRepository('SdzBlogBundle:Article')-
>find($id_article);

    // On modifie l'URL de l'image par exemple
    $article->getImage()->setUrl('test.png');

    // On n'a pas besoin de persister notre article (si vous le
    faites, aucune erreur n'est déclenchée, Doctrine l'ignore)
    // Rappelez-vous, il l'est automatiquement car on l'a récupéré
    depuis Doctrine

    // Pas non plus besoin de persister l'image ici, car elle est
    également récupérée par Doctrine

    // On déclenche la modification
    $em->flush();

    return new Response('OK');
}

```

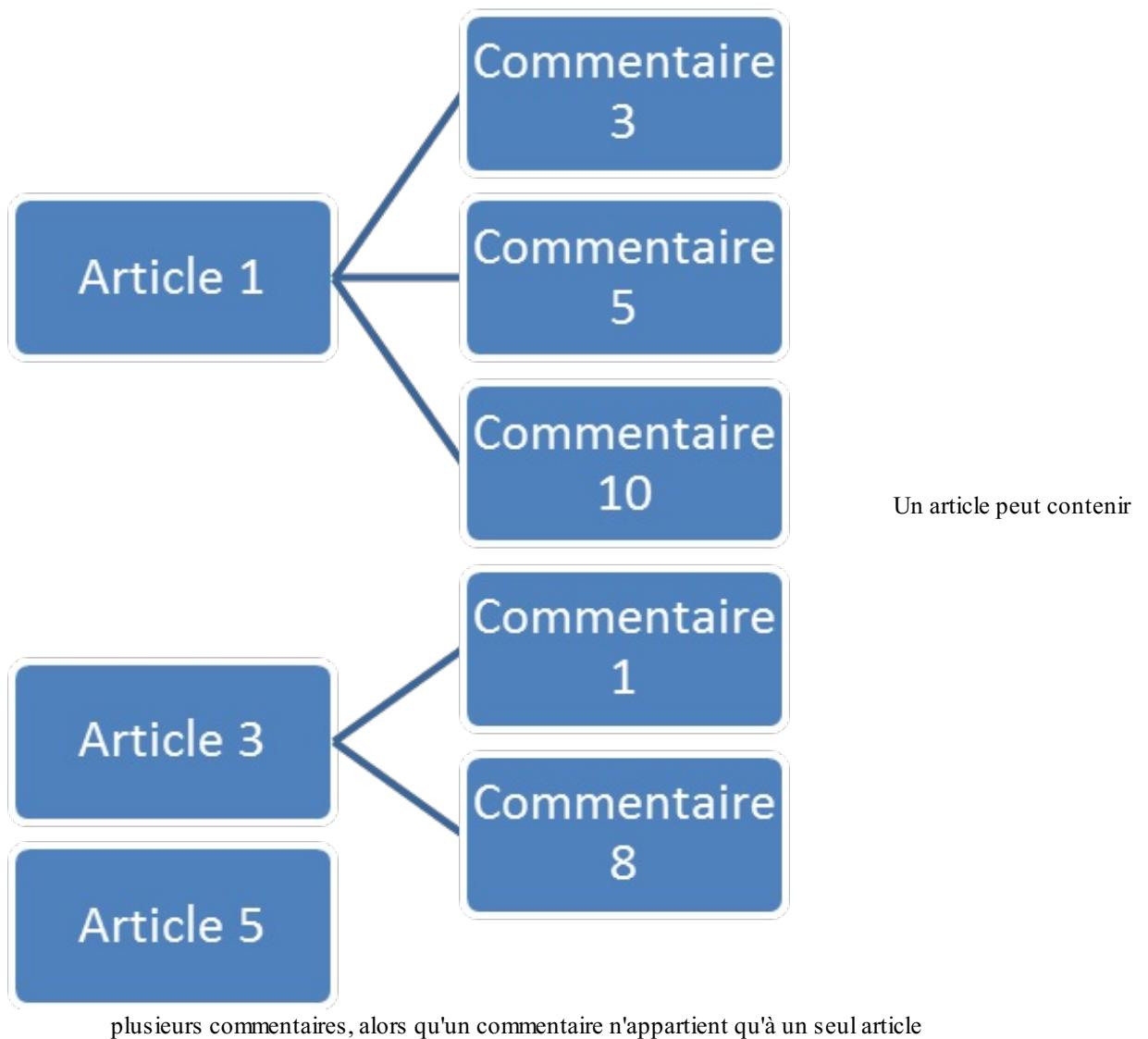
Le code parle de lui-même : gérer une relation est vraiment aisé avec Doctrine !

Relation Many-To-One

Présentation

La relation *Many-To-One*, ou **n..1**, est assez classique également. Elle correspond, comme son nom l'indique, à une relation qui permet à une entité A d'avoir une relation avec plusieurs entités B.

Pour illustrer cette relation dans le cadre de notre blog, nous allons créer une entité *Commentaire*. L'idée est de pouvoir ajouter *plusieurs commentaires à un article*, et que *chaque commentaire ne soit lié qu'à un seul article*. Nous avons ainsi plusieurs commentaires (*Many*) à lier (*To*) à un seul article (*One*). La figure suivante schématisé tout cela.



Comme précédemment, pour être sûrs qu'on parle bien de la même chose, créez cette entité `Commentaire` avec au moins les attributs `auteur`, `contenu` et `date`. Voici la mienne :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Sdz\BlogBundle\Entity\Commentaire
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CommentaireRepository")
 */
class Commentaire
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
}
```

```
    private $id;  
  
    /**  
     * @var string $auteur  
     *  
     * @ORM\Column(name="auteur", type="string", length=255)  
     */  
    private $auteur;  
  
    /**  
     * @var text $contenu  
     *  
     * @ORM\Column(name="contenu", type="text")  
     */  
    private $contenu;  
  
    /**  
     * @var datetime $date  
     *  
     * @ORM\Column(name="date", type="datetime")  
     */  
    private $date;  
  
    public function __construct()  
    {  
        $this->date = new \Datetime();  
    }  
  
    /**  
     * Get id  
     *  
     * @return integer  
     */  
    public function getId()  
    {  
        return $this->id;  
    }  
  
    /**  
     * Set auteur  
     *  
     * @param string $auteur  
     */  
    public function setAuteur($auteur)  
    {  
        $this->auteur = $auteur;  
    }  
  
    /**  
     * Get auteur  
     *  
     * @return string  
     */  
    public function getAuteur()  
    {  
        return $this->auteur;  
    }  
  
    /**  
     * Set contenu  
     *  
     * @param text $contenu  
     */  
    public function setContenu($contenu)  
    {  
        $this->contenu = $contenu;  
    }  
  
    /**  
     * Get contenu  
     */
```

```

/*
 * @return text
 */
public function getContenu()
{
    return $this->contenu;
}

/**
 * Set date
 *
 * @param \Datetime $date
 */
public function setDate(\Datetime $date)
{
    $this->date = $date;
}

/**
 * Get date
 *
 * @return \Datetime
 */
public function getDate()
{
    return $this->date;
}

```

Définition de la relation dans les entités

Annotation

Pour établir cette relation dans votre entité, la syntaxe est la suivante :

Entité propriétaire, **Commentaire** :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

/**
 * @ORM\Entity
 */
class Commentaire
{
    /**
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     * @ORM\JoinColumn(nullable=false)
     */
    private $article;

    // ...
}

```

Entité inverse, **Article** :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Article.php

```

```

/**
 * @ORM\Entity
 */
class Article
{
    // Nul besoin de rajouter de propriété, ici
    //
}

```

L'annotation à utiliser est tout simplement `ManyToOne`.

Première remarque : l'entité propriétaire pour cette relation est `Commentaire`, et non `Article`. Pourquoi ? Parce que, rappelez-vous, le propriétaire est celui qui contient la colonne référence. Ici, on aura bien une colonne `article_id` dans la table `commentaire`. En fait, de façon systématique, c'est le côté *Many* d'une relation *Many-To-One* qui est le propriétaire, vous n'avez pas le choix. Ici, on a plusieurs commentaires pour un seul article, le *Many* correspond aux commentaires, donc l'entité `Commentaire` est la propriétaire.

Deuxième remarque : j'ai volontairement ajouté l'annotation `JoinColumn` avec son attribut `nullable` à `false`, pour interdire la création d'un commentaire sans article. En effet, dans notre cas, un commentaire qui n'est rattaché à aucun article n'a pas de sens. Après, attention, il se peut très bien que dans votre application vous deviez laisser la possibilité au côté *Many* de la relation d'exister sans forcément être attaché à un côté *One*.



N'oubliez pas de mettre à jour la base de données avec la commande `doctrine:schema:update` !

Getter et setter

Ajoutons maintenant le getter et le setter correspondants dans l'entité propriétaire. Comme tout à l'heure, vous pouvez utiliser la méthode `php app/console doctrine:generate:entities SdzBlogBundle:Commentaire`, ou alors mettez ceux-là :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

/**
 * @ORM\Entity
 */
class Commentaire
{
    /**
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     * @ORM\JoinColumn(nullable=false)
     */
    private $article;

    // ... reste des attributs

    /**
     * Set article
     *
     * @param Sdz\BlogBundle\Entity\Article $article
     */
    public function setArticle(\Sdz\BlogBundle\Entity\Article
$article)
    {
        $this->article = $article;
    }

}

```

```

 * Get article
 *
 * @return Sdz\BlogBundle\Entity\Article
 */
public function getArticle()
{
    return $this->article;
}

// ... reste des getters et setters
}

```



Vous pouvez remarquer que, comme notre relation est obligatoire, il n'y a pas le « = null » dans le setArticle().

Exemple d'utilisation

La méthode pour gérer une relation *Many-To-One* n'est pas très différente de celle pour une relation *One-To-One*, voyez par vous-mêmes dans ces exemples.

Tout d'abord, pour ajouter un nouvel Article et ses Commentaires, modifions encore la méthode ajouterAction() de notre contrôleur :

Code : PHP

```

<?php
// src/Sdz/Bundle/Controller/BlogController.php

// N'oubliez pas ce use !
use Sdz\BlogBundle\Entity\Commentaire;

public function ajouterAction()
{
    // Création de l'entité Article
    $article = new Article();
    $article->setTitre('Mon dernier weekend');
    $article->setContenu("C'était vraiment super et on s'est bien
amusé.");
    $article->setAuteur('winzou');

    // Création d'un premier commentaire
    $commentaire1 = new Commentaire();
    $commentaire1->setAuteur('winzou');
    $commentaire1->setContenu('On veut les photos !');

    // Création d'un deuxième commentaire, par exemple
    $commentaire2 = new Commentaire();
    $commentaire2->setAuteur('Choupy');
    $commentaire2->setContenu('Les photos arrivent !');

    // On lie les commentaires à l'article
    $commentaire1->setArticle($article);
    $commentaire2->setArticle($article);

    // On récupère l'EntityManager
    $em = $this->getDoctrine()->getManager();

    // Étape 1 : On persiste les entités
    $em->persist($article);
    // Pour cette relation pas de cascade, car elle est définie
    // dans l'entité Commentaire et non Article
    // On doit donc tout persister à la main ici
    $em->persist($commentaire1);
}

```

```

$em->persist($commentaire2);

// Étape 2 : On déclenche l'enregistrement
$em->flush();

// ... reste de la méthode
}

```

Pour information, voici comment on pourrait modifier l'action `voirAction()` du contrôleur pour passer non seulement l'article à la vue, mais également ses commentaires (lignes 20, 21 et 26) :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function voirAction($id)
{
    // On récupère l'EntityManager
    $em = $this->getDoctrine()
        ->getManager();

    // On récupère l'entité correspondant à l'id $id
    $article = $em->getRepository('SdzBlogBundle:Article')
        ->find($id);

    if($article === null)
    {
        throw $this->createNotFoundException('Article[id='.$id.']'
inexistant.');
    }

    // On récupère la liste des commentaires
    $liste_commentaires = $em-
>getRepository('SdzBlogBundle:Commentaire')
        ->findAll();

    // Puis modifiez la ligne du render comme ceci, pour prendre en
    // compte l'article :
    return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
        'article'      => $article,
        'liste_commentaires' => $liste_commentaires
    ));
}

```

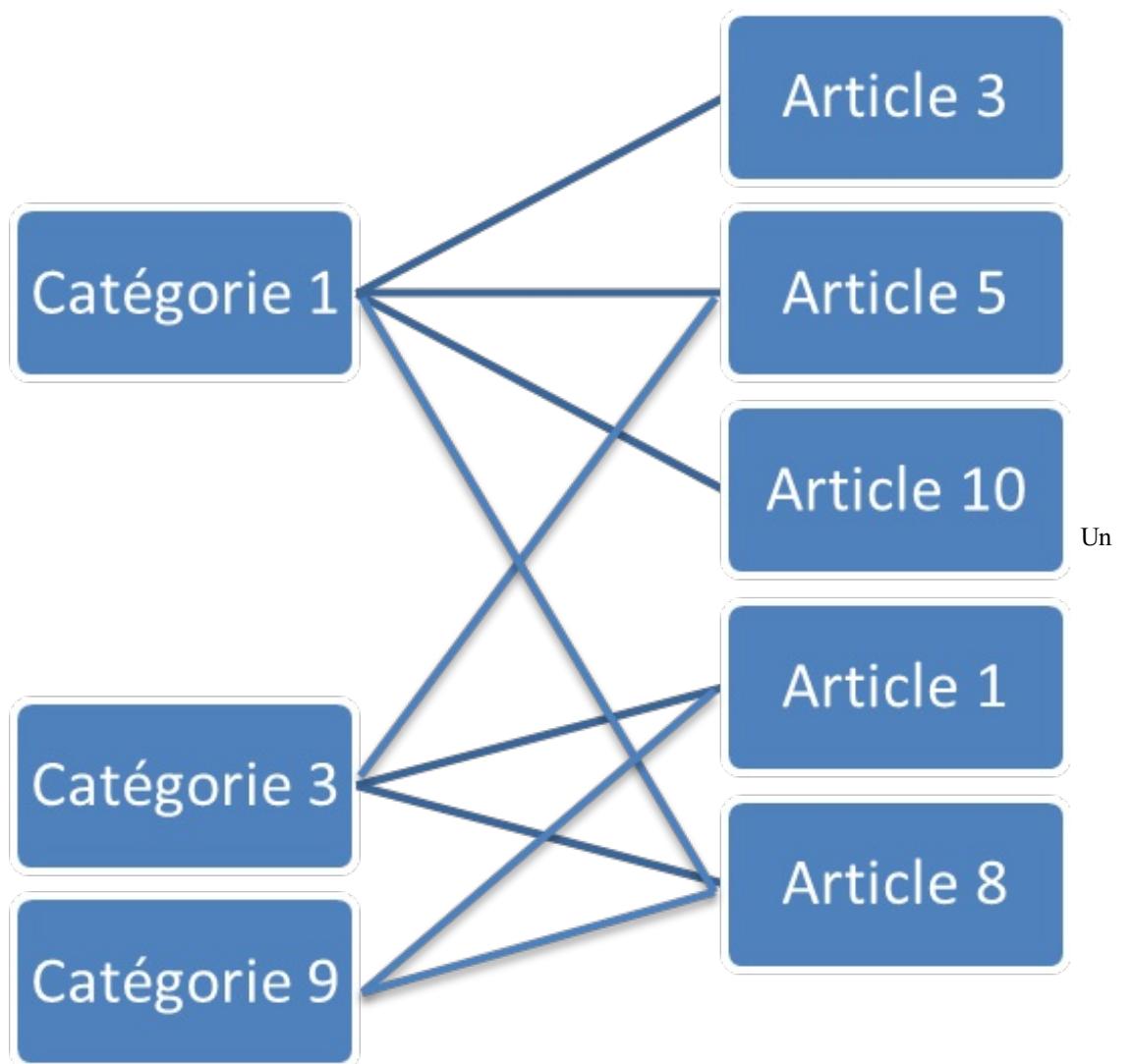
Ici vous pouvez voir qu'on a utilisé la méthode `findAll()`, qui récupère tous les commentaires et pas seulement ceux de l'article actuel. Il faudra bien sûr modifier ce comportement, nous le ferons dans le prochain chapitre, avec le repository qui permet de personnaliser nos requêtes. Et bien entendu, il faudrait adapter la vue si vous voulez afficher la liste des commentaires que nous venons de lui passer.

Relation Many-To-Many

Présentation

La relation *Many-To-Many*, ou **n.n**, correspond à une relation qui permet à plein d'objets d'être en relation avec plein d'autres !

Prenons l'exemple cette fois-ci des articles de notre blog, répartis dans des catégories. *Un article peut appartenir à plusieurs catégories. À l'inverse, une catégorie peut contenir plusieurs articles.* On a donc une relation *Many-To-Many* entre Article et Catégorie. La figure suivante schématisé tout cela.



Un article peut appartenir à plusieurs catégories et une catégorie peut contenir plusieurs articles

Cette relation est particulière dans le sens où Doctrine va devoir créer une table intermédiaire. En effet, avec la méthode traditionnelle en base de données, comment ferez-vous pour faire ce genre de relation ? Vous avez une table `article`, une autre table `categorie`, mais vous avez surtout besoin d'une table `article_categorie` qui fait la liaison entre les deux ! Cette table de liaison ne contient que deux colonnes : `article_id` et `categorie_id`. Cette table intermédiaire, vous ne la connaîtrez pas : elle n'apparaît pas dans nos entités, et c'est Doctrine qui la crée et qui la gère tout seul !

 Je vous ai parlé de cette table intermédiaire pour que vous compreniez comment Doctrine fonctionne. Cependant attention, nous sommes d'accord que vous devez totalement oublier cette notion de table intermédiaire lorsque vous manipulez des objets (les entités) ! J'insiste sur le fait que si vous voulez utiliser Doctrine, alors il faut le laisser gérer la base de données tout seul : vous utilisez des objets, lui utilise une base de données, chacun son travail.

Encore une fois, pour être sûrs que l'on parle bien de la même chose, créez cette entité `Categorie` avec au moins un attribut `nom`. Voici la mienne :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Categorie.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Sdz\BlogBundle\Entity\Categorie
 */

```

```

* @ORM\Table()
*
@ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CategorieRepository")
*/
class Categorie
{
    /**
 * @var integer $id
 *
 * @ORM\Column(name="id", type="integer")
 * @ORM\Id
 * @ORM\GeneratedValue(strategy="AUTO")
 */
    private $id;

    /**
 * @var string $nom
 *
 * @ORM\Column(name="nom", type="string", length=255)
 */
    private $nom;

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

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

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

```

Définition de la relation dans les entités

Annotation

Pour établir cette relation dans vos entités, la syntaxe est la suivante.

Entité propriétaire, Article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Categorie",
     * cascade={"persist"})
     */
    private $categories;

    // ...
}
```

Entité inverse, Categorie :**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Entity/Categorie.php

/**
 * @ORM\Entity
 */
class Categorie
{
    // Nul besoin d'ajouter une propriété, ici
    // ...
}
```

J'ai mis `Article` comme propriétaire de la relation. C'est un choix que vous pouvez faire comme bon vous semble, ici. Mais bon, récupérer les catégories d'un article se fera assez souvent, alors que récupérer les articles d'une catégorie moins. Et puis, pour récupérer les articles d'une catégorie, on aura sûrement besoin de personnaliser la requête, donc on le fera de toute façon depuis le `CategorieRepository`.

Getter et setters

Dans ce type de relation, il faut soigner un peu plus l'entité propriétaire. Tout d'abord, on a pour la première fois un attribut (ici `$categories`) qui contient une liste d'objets. C'est parce qu'il contient une liste d'objets qu'on a mis le nom de cet attribut au pluriel. Les listes d'objets avec Doctrine2 ne sont pas de simples tableaux, mais des `ArrayCollection`, il faudra donc définir l'attribut comme tel dans le constructeur. Un `ArrayCollection` est un objet utilisé par Doctrine2, qui a toutes les propriétés d'un tableau normal. Vous pouvez faire un `foreach` dessus, et le traiter comme n'importe quel tableau. Il dispose juste de quelques méthodes supplémentaires très pratiques, que nous verrons.

Ensuite, le getter est classique et s'appelle `getCategories()`. Par contre, c'est les setters qui vont différer un peu. En effet, `$categories` est une liste de catégories, mais au quotidien ce qu'on va faire c'est ajouter une à une des catégories à cette liste. Il nous faut donc une méthode `addCategorie()` (sans « s », on n'ajoute qu'une seule catégorie à la fois) et non une `setCategories()`. Du coup, il nous faut également une méthode pour supprimer une catégorie de la liste, que l'on appelle `removeCategorie()`.

Ajoutons maintenant le getter et les setters correspondants dans l'entité propriétaire, `Article`. Comme tout à l'heure, vous pouvez utiliser la méthode `php app/console doctrine:generate:entities SdzBlogBundle:Article`, ou alors reprendre ce code :

Code : PHP

```
<?php
```

```
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Categorie",
     * cascade={"persist"})
     */
    private $categories;

    // ... vos autres attributs

    // Comme la propriété $categories doit être un ArrayCollection,
    souvenez-vous
    // On doit la définir dans un constructeur :
    public function __construct()
    {
        // Si vous aviez déjà un constructeur, ajoutez juste cette
        ligne :
        $this->categories = new
    \Doctrine\Common\Collections\ArrayCollection();
    }

    /**
     * Add categories
     *
     * @param Sdz\BlogBundle\Entity\Categorie $categories
     */
    public function addCategorie(\Sdz\BlogBundle\Entity\Categorie
$categorie) // addCategorie sans « s » !
    {
        // Ici, on utilise l'ArrayCollection vraiment comme un tableau,
        avec la syntaxe []
        $this->categories[] = $categorie;
    }

    /**
     * Remove categories
     *
     * @param Sdz\BlogBundle\Entity\Categorie $categories
     */
    public function removeCategorie(\Sdz\BlogBundle\Entity\Categorie
$categorie) // removeCategorie sans « s » !
    {
        // Ici on utilise une méthode de l'ArrayCollection, pour
        supprimer la catégorie en argument
        $this->categories->removeElement($categorie);
    }

    /**
     * Get categories
     *
     * @return Doctrine\Common\Collections\Collection
     */
    public function getCategories() // Notez le « s », on récupère
une liste de catégories ici !
    {
        return $this->categories;
    }

    // ... vos autres getters/setters
}
```



N'oubliez pas de mettre à jour la base de données avec la commande `doctrine:schema:update`.



Remplissons la base de données avec les fixtures

Avant de voir un exemple, j'aimerais vous faire ajouter quelques catégories en base de données, histoire d'avoir de quoi jouer avec. Pour cela, petit aparté, nous allons faire une fixture Doctrine ! Cela va nous permettre d'utiliser le bundle qu'on a installé lors du chapitre sur Composer.

Les fixtures Doctrine permettent de remplir la base de données avec un jeu de données que nous allons définir. Cela permet de pouvoir tester avec des vraies données, sans devoir les retaper à chaque fois : on les inscrit une fois pour toutes, et ensuite elles sont toutes insérées en base de données en une seule commande.

Tout d'abord, créons notre fichier de fixture pour l'entité `Categorie`. Les fixtures d'un bundle se trouvent dans le répertoire `DataFixtures/ORM` (ou `ODM` pour des documents). Voici à quoi ressemble notre fixture `Categories` :

Code : PHP

```
<?php
// src/Sdz/Bundle/DataFixtures/ORM/Categories.php

namespace Sdz\BlogBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Sdz\BlogBundle\Entity\Categorie;

class Categories implements FixtureInterface
{
    // Dans l'argument de la méthode load, l'objet $manager est
    // l'EntityManager
    public function load(ObjectManager $manager)
    {
        // Liste des noms de catégorie à ajouter
        $noms = array('Symfony2', 'Doctrine2', 'Tutorial', 'Évènement');

        foreach($noms as $i => $nom)
        {
            // On crée la catégorie
            $liste_categories[$i] = new Categorie();
            $liste_categories[$i]->setNom($nom);

            // On la persiste
            $manager->persist($liste_categories[$i]);
        }

        // On déclenche l'enregistrement
        $manager->flush();
    }
}
```

C'est tout ! On peut dès à présent insérer ces données dans la base de données. Voici donc la commande à exécuter :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:fixtures:load
Careful, database will be purged. Do you want to continue Y/N ?y
> purging database
> loading Sdz\BlogBundle\DataFixtures\ORM\Categories
```

Et voilà ! Les quatre catégories définies dans le fichier de fixture sont maintenant enregistrées en base de données, on va pouvoir s'en servir dans nos exemples. Par la suite, on rajoutera d'autres fichiers de fixture pour insérer d'autres entités en base de données : la commande les traitera tous l'un après l'autre.



Attention, comme vous avez pu le voir, l'exécution de la commande Doctrine pour insérer les fixtures vide totalement la base de données avant d'insérer les nouvelles données. Si vous voulez ajouter les fixtures en plus des données déjà présentes, il faut ajouter l'option `--append` à la commande précédente. Cependant, c'est rarement ce que vous voulez, car à la prochaine exécution des fixtures, vous allez insérer une nouvelle fois les mêmes catégories...

Exemples d'utilisation

Voici un exemple pour ajouter un article existant à plusieurs catégories existantes. Je vous propose de mettre ce code dans notre méthode `modifierAction()` par exemple :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

class BlogController extends Controller
{
    // ...

    // Ajout d'un article existant à plusieurs catégories existantes :
    public function modifierAction($id)
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
            ->getManager();

        // On récupère l'entité correspondant à l'id $id
        $article = $em->getRepository('SdzBlogBundle:Article')
            ->find($id);

        if ($article === null) {
            throw $this->createNotFoundException('Article[id='.$id.']'
inexistant.');
        }

        // On récupère toutes les catégories :
        $liste_categories = $em-
>getRepository('SdzBlogBundle:Categorie')
            ->findAll();

        // On boucle sur les catégories pour les lier à l'article
        foreach($liste_categories as $categorie)
        {
            $article->addCategorie($categorie);
        }

        // Inutile de persister l'article, on l'a récupéré avec
Doctrine

        // Étape 2 : On déclenche l'enregistrement
        $em->flush();

        return new Response('OK');
    }
}
```

Je vous mets un exemple concret d'application pour que vous puissiez vous représenter l'utilisation de la relation dans un vrai cas d'utilisation. Les seules lignes qui concernent vraiment l'utilisation de notre relation *Many-To-Many* sont les lignes 29 à 32 : la boucle sur les catégories pour ajouter chaque catégorie une à une à l'article en question.

Voici un autre exemple pour enlever toutes les catégories d'un article. Modifions la méthode `supprimerAction()` pour l'occasion :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

class BlogController extends Controller
{
    // ...

    // Suppression des catégories d'un article :
    public function supprimerAction($id)
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
                    ->getManager();

        // On récupère l'entité correspondant à l'id $id
        $article = $em->getRepository('SdzBlogBundle:Article')
                        ->find($id);

        if ($article === null) {
            throw $this->createNotFoundException('Article[id='.$id.'] inexistant.');
        }

        // On récupère toutes les catégories :
        $liste_categories = $em-
            >getRepository('SdzBlogBundle:Categorie')
                ->findAll();

        // On enlève toutes ces catégories de l'article
        foreach($liste_categories as $categorie)
        {
            // On fait appel à la méthode removeCategorie() dont on a parlé plus haut
            // Attention ici, $categorie est bien une instance de Categorie, et pas seulement un id
            $article->removeCategorie($categorie);
        }

        // On n'a pas modifié les catégories : inutile de les persister
        // On a modifié la relation Article - Categorie
        // Il faudrait persister l'entité propriétaire pour persister la relation
        // Or l'article a été récupéré depuis Doctrine, inutile de le persister

        // On déclenche la modification
        $em->flush();
    }

    return new Response('OK');
}
}
```

Encore une fois, je vous ai mis un code complet, mais ce qui nous intéresse dans le cadre de la relation, ce ne sont que les lignes 30 à 35.

Enfin, voici un dernier exemple pour afficher les catégories d'un article dans la vue. Regardez plus particulièrement les lignes 21 à 27 :

Code : HTML & Django

```
{# src/Sdz/Bundle/Resources/views/Blog/voir.html.twig #}

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

{%
    block title %}
    Lecture d'un article - {{ parent() }}
{%
    endblock %}

{%
    block sdzblog_body %}

<h2>
    {# On vérifie qu'une image est bien associée à l'article #}
    {%
        if article.image is not null %}
        
<i>Par {{ article.auteur }}, le {{ article.date|date('d/m/Y') }}</i>

{%
    if article.categories.count > 0 %}
    - Catégories :
    {%
        for categorie in article.categories %}
        {{ categorie.nom }}
        {%
            if not loop.last %} - {%
            endif %}
    {%
        endfor %}
    {%
        endif %}

<div class="well">
    {{ article.contenu }}
</div>

<p>
    <a href="{{ path('sdzblog_accueil') }}" class="btn">
        <i class="icon-chevron-left"></i>
        Retour à la liste
    </a>
    <a href="{{ path('sdzblog_modifier', {'id': article.id}) }}" class="btn">
        <i class="icon-edit"></i>
        Modifier l'article
    </a>
    <a href="{{ path('sdzblog_supprimer', {'id': article.id}) }}" class="btn">
        <i class="icon-trash"></i>
        Supprimer l'article
    </a>
</p>

{%
    endblock %}
```

Vous voyez qu'on accède aux catégories d'un article grâce à l'attribut `categories` de l'article, tout simplement. En Twig, cela signifie `{{ article.categories }}`, en PHP on ferait `$article->getCategories()`. Il suffit ensuite de parcourir ce tableau grâce à une boucle, et d'en faire ce que l'on veut.

Relation Many-To-Many avec attributs

Présentation

La relation *Many-To-Many* qu'on vient de voir peut suffire dans bien des cas, mais elle est en fait souvent incomplète pour les

besoins d'une application.

Pour illustrer ce manque, rien de tel qu'un exemple : considérons l'entité `Produit` d'un site e-commerce ainsi que l'entité `Commande`. Une commande contient plusieurs produits, et bien entendu un même produit peut être dans différentes commandes. On a donc bien une relation *Many-To-Many*. Voyez-vous le manque ? Lorsqu'un utilisateur ajoute un produit à une commande, où met-on la quantité de ce produit qu'il veut ? Si je veux 3 exemplaires de Harry Potter, où mettre cette quantité ? Dans l'entité `Commande` ? Non cela n'a pas de sens. Dans l'entité `Produit` ? Non, cela n'a pas de sens non plus. Cette quantité est un attribut de la relation qui existe entre `Produit` et `Commande`, et non un attribut de `Produit` ou de `Commande`.

Il n'y a pas de moyen simple de gérer les attributs d'une relation avec Doctrine. Pour cela, il faut esquerir en créant simplement une entité intermédiaire qui va représenter la relation, appelons-la `CommandeProduit`. Et c'est dans cette entité que l'on mettra les attributs de relation, comme notre quantité. Ensuite, il faut bien entendu mettre en relation cette entité intermédiaire avec les deux autres entités d'origine, `Commande` et `Produit`. Pour cela, il faut logiquement faire : `Commande One-To-Many CommandeProduit Many-To-One Produit`. En effet, une commande (*One*) peut avoir plusieurs relations avec des produits (*Many*), plusieurs `CommandeProduit`, donc ! La relation est symétrique pour les produits.

Attention, dans le titre de cette section, j'ai parlé de la relation *Many-To-Many avec attributs*, mais il s'agit bien en fait de deux relations *Many-To-One* des plus normales, soyons d'accord. On ne va donc rien apprendre dans ce prochain paragraphe, car on sait déjà faire une *Many-To-One*, mais c'est une astuce qu'il faut bien connaître et savoir utiliser, donc prenons le temps de bien la comprendre.

J'ai pris l'exemple de produits et de commandes, car c'est plus intuitif pour comprendre l'enjeu et l'utilité de cette relation. Cependant, pour rester dans le cadre de notre blog, on va faire une relation entre des `Article` et des `Compétence`, et l'attribut de la relation sera le niveau. L'idée est de pouvoir afficher sur chaque article la liste des compétences utilisées (Symfony2, Doctrine2, Formulaire, etc.) avec le niveau dans chaque compétence (Débutant, Avisé et Expert). On a alors l'analogie suivante :

- `Article <=> Commande`
- `ArticleCompetence <=> Commande_Produit`
- `Competence <=> Produit`

Et donc : `Article One-To-Many ArticleCompetence Many-To-One Competence`.

Pour cela, créez d'abord cette entité `Compétence`, avec au moins un attribut nom. Voici la mienne :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Competence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Sdz\BlogBundle\Entity\Competence
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CompetenceRepository")
 */
class Competence
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $nom
     *
     * @ORM\Column(name="nom", type="string", length=255)
     */
    private $nom;
}
```

```

/*
    private $nom;

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

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

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

```

Définition de la relation dans les entités

Annotation

Tout d'abord, on va créer notre entité de relation (notre ArticleCompetence) comme ceci :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/ArticleCompetence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class ArticleCompetence
{
    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     */
    private $article;

    /**
     * @ORM\Id
     */

```

```

* @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Competence")
*/
private $competence;

/**
* @ORM\Column()
*/
private $niveau; // Ici j'ai un attribut de relation « niveau »

// ... vous pouvez ajouter d'autres attributs bien entendu
}

```

Comme les côtés *Many* des deux relations *Many-To-One* sont dans ArticleCompetence, cette entité est l'entité propriétaire des deux relations.



Et ces `@ORM\Id` ? Pourquoi y en a-t-il deux et qu'est-ce qu'ils viennent faire ici ?

Très bonne question. Comme toute entité, notre ArticleCompetence se doit d'avoir un identifiant. C'est obligatoire pour que Doctrine puisse la gérer par la suite. Depuis le début, nous avons ajouté un attribut `id` qui était en auto-incrémentation, on ne s'en occupait pas trop donc. Ici c'est différent, comme une ArticleCompetence correspond à un unique couple Article/Competence (pour chaque couple Article/Competence, il n'y a qu'une seule ArticleCompetence), on peut se servir de ces deux attributs pour former l'identifiant de cette entité.

Pour cela, il suffit de définir `@ORM\Id` sur les deux colonnes, et Doctrine saura mettre une clé primaire sur ces deux colonnes, puis les gérer comme n'importe quel autre identifiant. Encore une fois, merci Doctrine !



Mais, avec une relation unidirectionnelle, on ne pourra pas faire `$article->getArticleCompetence()` pour récupérer les ArticleCompetence et donc les compétences ? Ni l'inverse depuis `$competence` ?

En effet, et c'est pourquoi la prochaine section de ce chapitre traite des relations bidirectionnelles ! En attendant, pour notre relation *One-To-Many-To-One*, continuons simplement sur une relation unidirectionnelle.

Sachez quand même que vous pouvez éviter une relation bidirectionnelle ici en utilisant simplement la méthode `findByCommande ($commande->getId())` (pour récupérer les produits d'une commande) ou `findByProduit ($produit->getId())` (pour l'inverse) du repository `CommandeProduitRepository`.

L'intérêt de la bidirectionnelle ici est lorsque vous voulez afficher une liste des commandes avec leurs produits. Dans la boucle sur les commandes, vous n'allez pas faire appel à une méthode du repository qui va générer une requête par boucle, il faudra passer par un `$commande->getCommandeProduits()`. Nous le verrons plus loin dans ce chapitre.



N'oubliez pas de mettre à jour votre base de données en exécutant la commande `doctrine:schema:update`.

Getters et setters

Comme d'habitude les getters et setters doivent se définir dans l'entité propriétaire. Ici, rappelez-vous, nous sommes en présence de deux relations *Many-To-One* dont la propriétaire est l'entité ArticleCompetence. Nous avons donc deux getters et deux setters classiques à écrire. Vous pouvez les générer avec la commande

`doctrine:generate:entities SdzBlogBundle:ArticleCompetence`, ou mettre le code suivant :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/ArticleCompetence.php

namespace Sdz\BlogBundle\Entity;

```

```

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class ArticleCompetence
{
    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     */
    private $article;

    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Competence")
     */
    private $competence;

    /**
     * @ORM\Column()
     */
    private $niveau; // Ici j'ai un attribut de relation « niveau »

    // ... les autres attributs

    // Getter et setter pour l'entité Article
    public function setArticle(\Sdz\BlogBundle\Entity\Article $article)
    {
        $this->article = $article;
    }
    public function getArticle()
    {
        return $this->article;
    }

    // Getter et setter pour l'entité Competence
    public function setCompetence(\Sdz\BlogBundle\Entity\Competence $competence)
    {
        $this->competence = $competence;
    }
    public function getCompetence()
    {
        return $this->competence;
    }

    // On définit le getter/setter de l'attribut « niveau »
    public function setNiveau($niveau)
    {
        $this->niveau = $niveau;
    }
    public function getNiveau()
    {
        return $this->niveau;
    }

    // ... les autres getters/setters si vous en avez
}

```

Remplissons la base de données

Comme précédemment, on va d'abord ajouter des compétences en base de données grâce aux fixtures. Pour faire une nouvelle fixture, il suffit de créer un nouveau fichier dans le répertoire DataFixtures/ORM dans le bundle. Je vous invite à créer le

fichier `src/Sdz/BlogBundle/DataFixtures/ORM/Competences.php` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/DataFixtures/ORM/Competences.php

namespace Sdz\BlogBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Sdz\BlogBundle\Entity\Competence;

class Competences implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        // Liste des noms de compétences à ajouter
        $noms = array('Doctrine', 'Formulaire', 'Twig');

        foreach($noms as $i => $nom)
        {
            // On crée la compétence
            $liste_competences[$i] = new Competence();
            $liste_competences[$i]->setNom($nom);

            // On la persiste
            $manager->persist($liste_competences[$i]);
        }

        // On déclenche l'enregistrement
        $manager->flush();
    }
}
```

Et maintenant, on peut exécuter la commande :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:fixtures:load
Careful, database will be purged. Do you want to continue Y/N ?y
> purging database
> loading Sdz\BlogBundle\DataFixtures\ORM\Categories
> loading Sdz\BlogBundle\DataFixtures\ORM\Competences
```

Vous pouvez voir qu'après avoir tout vidé Doctrine a inséré les fixtures Categories puis nos fixtures Competences. Tout est prêt !

Exemple d'utilisation

La manipulation des entités dans une telle relation est un peu plus compliquée, surtout sans la bidirectionnalité. Mais on peut tout de même s'en sortir. Tout d'abord, voici un exemple pour créer un nouvel article contenant plusieurs compétences ; mettons ce code dans la méthode `ajouterAction()` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// N'oubliez pas ce use évidemment
```

```

use Sdz\BlogBundle\Entity\ArticleCompetence;

class BlogController extends Controller
{
    // ...

    public function ajouterAction()
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
            ->getManager();

        // Création de l'entité Article
        $article = new Article();
        $article->setTitre('Mon dernier weekend');
        $article->setContenu("C'était vraiment super et on s'est bien
        amusé.");
        $article->setAuteur('winzou');

        // Dans ce cas, on doit créer effectivement l'article en bdd
        // pour lui assigner un id
        // On doit faire cela pour pouvoir enregistrer les
        ArticleCompetence par la suite
        $em->persist($article);
        $em->flush(); // Maintenant, $article a un id défini

        // Les compétences existent déjà, on les récupère depuis la bdd
        $liste_competences = $em-
        >getRepository('SdzBlogBundle:Competence')
            ->findAll(); // Pour l'exemple, notre
        Article contient toutes les Competences

        // Pour chaque compétence
        foreach($liste_competences as $i => $competence)
        {
            // On crée une nouvelle « relation entre 1 article et 1
            compétence »
            $articleCompetence[$i] = new ArticleCompetence;

            // On la lie à l'article, qui est ici toujours le même
            $articleCompetence[$i]->setArticle($article);
            // On la lie à la compétence, qui change ici dans la boucle
        foreach
            $articleCompetence[$i]->setCompetence($competence);

            // Arbitrairement, on dit que chaque compétence est requise
            au niveau 'Expert'
            $articleCompetence[$i]->setNiveau('Expert');

            // Et bien sûr, on persiste cette entité de relation,
            propriétaire des deux autres relations
            $em->persist($articleCompetence[$i]);
        }

        // On déclenche l'enregistrement
        $em->flush();

        // ... reste de la méthode
    }
}

```

Et voici un autre exemple pour récupérer les compétences et leur niveau à partir d'un article donné. Je vous propose de modifier la méthode `voirAction()` :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

class BlogController extends Controller
{
    // ...

    public function voirAction($id)
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
            ->getManager();

        // On récupère l'entité correspondant à l'id $id
        $article = $em->getRepository('SdzBlogBundle:Article')
            ->find($id);

        if ($article === null) {
            throw $this->createNotFoundException('Article[id='.$id.'] inexistant.');
        }

        // On récupère les articleCompetence pour l'article $article
        $liste_articleCompetence = $em-
>getRepository('SdzBlogBundle:ArticleCompetence')
            ->findByArticle($article->getId());

        // Puis modifiez la ligne du render comme ceci, pour prendre en compte les articleCompetence :
        return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
            'article'      => $article,
            'liste_articleCompetence' => $liste_articleCompetence,
            // ... et évidemment les autres variables que vous pouvez avoir
        ));
    }
}

```

Et le code de la vue correspondante :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

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

{%
    block title %
        Lecture d'un article - {{ parent() }}
{%
    endblock %}

{%
    block sdzblog_body %

        <h2>
            {# On vérifie qu'une image est bien associée à l'article #}
            {%
                if article.image is not null %}
                    
            {%
                endif %}

            {{ article.titre }}
        </h2>
        <i>Par {{ article.auteur }}, le {{ article.date|date('d/m/Y') }}</i>

        {%
            if article.categories.count > 0 %}
                - Catégories :
                {%
                    for categorie in article.categories %}

```

```

        {{ categorie.nom }}
        {%- if not loop.last %} - {{ endif %}
    {%- endfor %}
{%- endif %}

<div class="well">
    {{ article.contenu }}
</div>

{%- if liste_articleCompetence|length > 0 %}
<div>
    Compétences utilisées dans cet article :
    <ul>
        {%- for articleCompetence in liste_articleCompetence %}
        <li>{{ articleCompetence.competence.nom }} : {{ articleCompetence.niveau }}</li>
        {%- endfor %}
    </ul>
</div>
{%- endif %}

<p>
    <a href="{{ path('sdzblog_accueil') }}" class="btn">
        <i class="icon-chevron-left"></i>
        Retour à la liste
    </a>
    <a href="{{ path('sdzblog_modifier', {'id': article.id}) }}" class="btn">
        <i class="icon-edit"></i>
        Modifier l'article
    </a>
    <a href="{{ path('sdzblog_supprimer', {'id': article.id}) }}" class="btn">
        <i class="icon-trash"></i>
        Supprimer l'article
    </a>
</p>

{%- endblock %}

```

C'est un exemple simple bien sûr. 😊

Attention, dans cet exemple, la méthode `findByArticle()` utilisée dans le contrôleur ne sélectionne que les `ArticleCompetence`. Donc, lorsque dans la boucle dans la vue on fait `{{ articleCompetence.competence }}`, en réalité Doctrine va effectuer une requête pour récupérer la `Competence` associée à cette `ArticleCompetence`. C'est bien sûr une horreur, car il va faire une requête... par itération dans le `for` ! Si vous avez 20 compétences attachées à l'article, cela ferait 20 requêtes : inimaginable.

Pour charger les `Competence` en même temps que les `ArticleCompetence` dans le contrôleur, et ainsi ne plus faire de requête dans la boucle, il faut faire une méthode à nous dans le repository de `ArticleCompetence`. On voit tout cela dans le chapitre suivant dédié aux repositories. N'utilisez donc jamais cette technique, attendez le prochain chapitre ! La seule différence dans le contrôleur sera d'utiliser une autre méthode que `findByArticle()`, et la vue ne changera même pas.

Les relations bidirectionnelles

Présentation

Vous avez vu que jusqu'ici nous n'avons jamais modifié l'entité inverse d'une relation, mais seulement l'entité propriétaire. Toutes les relations que l'on vient de faire sont donc des relations unidirectionnelles.

Leur avantage est de définir la relation d'une façon très simple. Mais l'inconvénient est de ne pas pouvoir récupérer l'entité propriétaire depuis l'entité inverse, le fameux `<?php $entiteInverse->getEntiteProprietaire()` (pour nous, `<?php $article->getCommentaires()` par exemple). Je dis inconvénient, mais vous avez pu constater que cela ne nous a pas du tout empêché de faire ce qu'on voulait ! À chaque fois, on a réussi à ajouter, lister, modifier nos entités et leurs relations.

Mais dans certains cas, avoir une relation bidirectionnelle est bien utile. Nous allons les voir rapidement dans cette section.

Sachez que la documentation l'explique également très bien : vous pourrez vous renseigner sur le chapitre sur la création des relations, puis celui sur leur utilisation.

Définition de la relation dans les entités

Pour étudier la définition d'une relation bidirectionnelle, nous allons étudier une relation *Many-To-One*. Souvenez-vous bien de cette relation, dans sa version unidirectionnelle, pour pouvoir attaquer sa version bidirectionnelle dans les meilleures conditions.

Nous allons ici construire une relation bidirectionnelle de type *Many-To-One*, basée sur notre exemple Article-Commentaire. Mais la méthode est exactement la même pour les relations de type *One-To-One* ou *Many-To-Many*.



Je pars du principe que vous avez déjà une relation unidirectionnelle fonctionnelle. Si ce n'est pas votre cas, mettez-la en place avant de lire la suite du paragraphe, car nous n'allons voir que les ajouts à faire.

Annotation

Alors, attaquons la gestion d'une relation bidirectionnelle. L'objectif de cette relation est de rendre possible l'accès à l'entité propriétaire depuis l'entité inverse. Avec une unidirectionnelle, cela n'est pas possible, car on n'ajoute pas d'attribut dans l'entité inverse, ce qui signifie que l'entité inverse ne sait même pas qu'elle fait partie d'une relation.

La première étape consiste donc à rajouter un attribut, et son annotation, à notre entité inverse Article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\OneToMany(targetEntity="Sdz\BlogBundle\Entity\Commentaire",
     * mappedBy="article")
    */
    private $commentaires; // Ici commentaires prend un « s », car un
                           // article a plusieurs commentaires !

    // ...
}
```

Bien entendu, je vous dois des explications sur ce que l'on vient de faire.

Commençons par l'annotation. L'inverse d'un *Many-To-One* est... un *One-To-Many*, tout simplement ! Il faut donc utiliser l'annotation *One-To-Many* dans l'entité inverse. Je rappelle que le propriétaire d'une relation *Many-To-One* est toujours le côté *Many*, donc, lorsque vous voyez l'annotation *Many-To-One*, vous êtes forcément du côté propriétaire. Ici on a un *One-To-Many*, on est bien du côté inverse.

Ensuite, les paramètres de cette annotation. Le *targetEntity* est évident, il s'agit toujours de l'entité à l'autre bout de la relation, ici notre entité Commentaire. Le *mappedBy* correspond, lui, à l'attribut de l'entité propriétaire (Commentaire) qui pointe vers l'entité inverse (Article) : c'est le *private \$article* de l'entité Commentaire. Il faut le renseigner pour que l'entité inverse soit au courant des caractéristiques de la relation : celles-ci sont définies dans l'annotation de l'entité propriétaire.

Il faut également adapter l'entité propriétaire, pour lui dire que maintenant la relation est de type bidirectionnelle et non plus unidirectionnelle. Pour cela, il faut rajouter le paramètre *inversedBy* dans l'annotation *Many-To-One* :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

/**
 * @ORM\Entity
 */
class Commentaire
{
    /**
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article",
     * inverseBy="commentaires")
     * @ORM\JoinColumn(nullable=false)
     */
    private $article;

    // ...
}
```

Ici, nous avons seulement rajouté le paramètre `inverseBy`. Il correspond au symétrique du `mappedBy`, c'est-à-dire à l'attribut de l'entité inverse (`Article`) qui pointe vers l'entité propriétaire (`Commentaire`). C'est donc l'attribut `commentaires`.

Tout est bon côté annotation, maintenant il faut également ajouter les getters et setters dans l'entité inverse bien entendu.

Getters et setters

On part d'une relation unidirectionnelle fonctionnelle, donc les getters et setters de l'entité propriétaire sont bien définis.

Dans un premier temps, ajoutons assez logiquement le getter et le setter dans l'entité inverse. On vient de lui ajouter un attribut, il est normal que le getter et le setter aillent de paire. Comme nous sommes du côté *One* d'un *One-To-Many*, l'attribut `commentaires` est un `ArrayCollection`. C'est donc un `addCommentaire` / `removeCommentaire` / `getCommentaires` qu'il nous faut. Encore une fois, vous pouvez le générer avec `doctrine:generate:entities SdzBlogBundle:Article`, ou alors vous pouvez mettre ce code :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\OneToMany(targetEntity="Sdz\BlogBundle\Entity\Commentaire",
     * mappedBy="article")
     */
    private $commentaires; // Ici commentaires prend un « s », car un article a plusieurs commentaires !

    // ... vos autres attributs

    public function __construct()
    {
        // Rappelez-vous, on a un attribut qui doit contenir un
        // ArrayCollection, on doit l'initialiser dans le constructeur
        $this->commentaires = new
        \Doctrine\Common\Collections\ArrayCollection();
    }

    public function addCommentaire(\Sdz\BlogBundle\Entity\Commentaire
$commentaire)
```

```

    {
        $this->commentaires[] = $commentaire;
        return $this;
    }

    public function
removeCommentaire(\Sdz\BlogBundle\Entity\Commentaire $commentaire)
{
    $this->commentaires->removeElement($commentaire);
}

public function getCommentaires()
{
    return $this->commentaires;
}

// ...
}

```

Maintenant, il faut nous rendre compte d'un petit manque. Voici une petite problématique, lisez bien ce code :

Code : PHP

```

<?php
// Création des entités
$article = new Article;
$commentaire = new Commentaire;

// On lie le commentaire à l'article
$article->addCommentaire($commentaire);

```



Que retourne `$commentaire->getArticle()` ?

Rien ! En effet, pour qu'un `$commentaire->getArticle()` retourne un article, il faut d'abord le lui définir en appelant `$commentaire->setArticle($article)`, c'est logique !



Si vous ne voyez pas pourquoi Doctrine n'a pas rempli l'attribut `article` de l'objet `$commentaire`, il faut revenir aux fondamentaux. Vous êtes en train d'écrire du PHP, `$article` et `$commentaire` sont des objets PHP, tels qu'ils existaient bien avant la naissance de Doctrine. Pour que l'attribut `article` soit défini, il faut absolument faire appel au setter `setArticle()`, car c'est le seul qui accède à cet attribut (qui est en `private`). Dans le petit exemple que je vous ai mis, on n'a pas exécuté de fonction Doctrine : à aucun moment il n'a pu intervenir et éventuellement exécuter le `setArticle()`. 😊

C'est logique en soi, mais du coup dans notre code cela va être moins beau : il faut en effet lier le commentaire à l'article *et* l'article au commentaire. Comme ceci :

Code : PHP

```

<?php
// Création des entités
$article = new Article;
$commentaire = new Commentaire;

// On lie le commentaire à l'article
$article->addCommentaire($commentaire);

// On lie l'article au commentaire

```

```
$commentaire->setArticle($article);
```

Mais ces deux méthodes étant intimement liées, on doit en fait les imbriquer. En effet, laisser le code en l'état est possible, mais imaginez qu'un jour vous oubliez d'appeler l'une des deux méthodes ; votre code ne sera plus cohérent. Et un code non cohérent est un code qui a des risques de contenir des bugs. La bonne méthode est donc simplement de faire appel à l'une des méthodes depuis l'autre. Voici concrètement comme le faire en modifiant les setters dans l'une des deux entités :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article

/**
 * @ORM\Entity
 */
class Article
{
    // ...

    public function addCommentaire(\Sdz\BlogBundle\Entity\Commentaire
$commentaire)
    {
        $this->commentaires[] = $commentaire;
        $commentaires->setArticle($this); // On ajoute ceci
        return $this;
    }

    public function
removeCommentaire(\Sdz\BlogBundle\Entity\Commentaire $commentaire)
    {
        $this->commentaires->removeElement($commentaire);
        // Et si notre relation était facultative (nullable=true, ce
qui n'est pas notre cas ici attention) :
        // $commentaire->setArticle(null);
    }

    // ...
}
```

Notez qu'ici j'ai modifié un côté de la relation (l'inverse en l'occurrence), mais surtout pas les deux ! En effet, si `addCommentaire()` exécute `setArticle()`, qui exécute à son tour `addCommentaire()`, qui... etc. On se retrouve avec une boucle infinie.

Bref, l'important est de se prendre un côté (propriétaire ou inverse, cela n'a pas d'importance), et de l'utiliser. Par utiliser, j'entends que dans le reste du code (contrôleur, service, etc.) il faudra exécuter `$article->addCommentaire()` qui garde la cohérence entre les deux entités. Il ne faudra pas exécuter `$commentaire->setArticle()`, car lui ne garde pas la cohérence ! Retenez : *on modifie le setter d'un côté, et on utilise ensuite ce setter-là*. C'est simple, mais important à respecter.

Pour conclure

Le chapitre sur les relations Doctrine touche ici à sa fin.

Pour maîtriser les relations que nous venons d'apprendre, il faut vous entraîner à les créer et à les manipuler. N'hésitez donc pas à créer des entités d'entraînement, et à voir leur comportement dans les relations.

Si vous voulez plus d'informations sur les fixtures que l'on a rapidement abordées lors de ce chapitre, je vous invite à lire la page de la documentation du bundle : [http://symfony.com/fr/doc/current/bundle \[...\] le/index.html](http://symfony.com/fr/doc/current/bundle [...] le/index.html)

Rendez-vous au prochain chapitre pour apprendre à récupérer les entités depuis la base de données à votre guise, grâce aux repositories !

En résumé

- Les relations Doctrine révèlent toute la puissance de l'ORM ;
- Dans une relation entre deux entités, l'une est propriétaire de la relation et l'autre est inverse. Cette notion est purement technique ;
- Une relation est dite unidirectionnelle si l'entité inverse n'a pas d'attribut la liant à l'entité propriétaire. On met en place une relation bidirectionnelle lorsqu'on a besoin de cet attribut dans l'entité inverse (ce qui arrivera pour certains formulaires, etc.).

Récupérer ses entités avec Doctrine2

L'une des principales fonctions de la couche Modèle dans une application MVC, c'est la récupération des données. Récupérer des données n'est pas toujours évident, surtout lorsqu'on veut récupérer seulement certaines données, les classer selon des critères, etc. Tout cela se fait grâce aux *repositories*, que nous étudions dans ce chapitre. Bonne lecture !

Le rôle des repositories

On s'est déjà rapidement servi de quelques repositories, donc vous devriez sentir leur utilité, mais il est temps de théoriser un peu.

Définition

Un *repository* centralise tout ce qui touche à la récupération de vos entités. Concrètement donc, vous ne devez pas faire la moindre requête SQL ailleurs que dans un repository, c'est la règle. On va donc y construire des méthodes pour récupérer une entité par son id, pour récupérer une liste d'entités suivant un critère spécifique, etc. Bref, à chaque fois que vous devez récupérer des entités dans votre base de données, vous utiliserez le repository de l'entité correspondante.

Rappelez-vous, il existe un repository par entité. Cela permet de bien organiser son code. Bien sûr, cela n'empêche pas qu'un repository utilise plusieurs types d'entité, dans le cas d'une jointure par exemple.

Les repositories ne fonctionnent pas par magie, ils utilisent en réalité directement l'EntityManager pour faire leur travail. Vous le verrez, parfois nous ferons directement appel à l'EntityManager depuis des méthodes du repository.

Les méthodes de récupération des entités

Depuis un repository, il existe deux moyens de récupérer les entités : en utilisant du DQL et en utilisant le QueryBuilder.

Le Doctrine Query Language (DQL)

Le DQL n'est rien d'autre que du SQL adapté à la vision par objets que Doctrine utilise. Il s'agit donc de faire ce qu'on a l'habitude de faire, des requêtes textuelles comme celle-ci par exemple :

Code : SQL

```
SELECT a FROM SdzBlogBundle:Article a
```

Vous venez de voir votre première requête DQL. Retenez le principe : avec une requête qui n'est rien d'autre que du texte, on effectue le traitement voulu.

Le QueryBuilder

Le QueryBuilder est un moyen plus nouveau. Comme son nom l'indique, il sert à construire une requête, par étape. Si l'intérêt n'est pas évident au début, son utilisation se révèle vraiment pratique ! Voici la même requête que précédemment, mais en utilisant le QueryBuilder :

Code : PHP

```
<?php
$QueryBuilder
    ->select('a')
    ->from('SdzBlogBundle:Article', 'a');
```

Un des avantages est qu'il est possible de construire la requête en plusieurs fois. Ainsi, vous pouvez développer une méthode qui rajoute une condition à une requête, par exemple pour sélectionner tous les membres actifs (qui se sont connectés depuis moins d'un mois par exemple). Comme cette condition risque de servir souvent, dans plusieurs requêtes, auparavant vous deviez la réécrire à chaque fois. Avec le QueryBuilder, vous pourrez faire appel à la même méthode, sans réécrire la condition. Pas de

panique, on verra des exemples dans la suite du chapitre !

Les méthodes de récupération de base

Définition

Vos repositories héritent de la classe `Doctrine\ORM\EntityRepository`, qui propose déjà quelques méthodes très utiles pour récupérer des entités. Ce sont ces méthodes-là que nous allons voir ici.

Les méthodes normales

Il existe quatre méthodes, que voici (tous les exemples sont effectués depuis un contrôleur).

`find($id)`

La méthode `find($id)` récupère tout simplement l'entité correspondant à l'id `$id`. Dans le cas de notre `ArticleRepository`, elle retourne une instance d'`Article`. Exemple :

Code : PHP

```
<?php
$repository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');

$article_5 = $repository->find(5);
// $article_5 est une instance de Sdz\BlogBundle\Entity\Article
```

`findAll()`

La méthode `findAll()` retourne toutes les entités. Le format du retour est un simple `Array`, que vous pouvez parcourir (avec un `foreach` par exemple) pour utiliser les objets qu'il contient. Exemple :

Code : PHP

```
<?php
$repository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');

$listeArticles = $repository->findAll();

foreach($listeArticles as $article)
{
    // $article est une instance de Article
    echo $article->getContenu();
}
```

Ou dans une vue Twig, si l'on a passé la variable `$listeArticles` au template :

Code : HTML

```
<ul>
    {# for article in listeArticles #}
        <li>{{ article.contenu }}</li>
    {# endfor #}
</ul>
```

findBy()

La méthode `findBy()` est un peu plus intéressante. Comme `findAll()`, elle permet de retourner une liste d'entités, sauf qu'elle est capable d'effectuer un filtre pour ne retourner que les entités correspondant à un critère. Elle peut aussi trier les entités, et même n'en récupérer qu'un certain nombre (pour une pagination).

La syntaxe est la suivante :

Code : PHP

```
<?php
$repository->findBy(array $ criteres, array $orderBy = null, $limite
= null, $offset = null);
```

Voici un exemple d'utilisation :

Code : PHP

```
<?php

getRepository = $this->getDoctrine()
->getManager()
->getRepository('SdzBlogBundle:Article');

$listeArticles = $repository->findBy(array('auteur' => 'winzou'),
array('date' => 'desc'),
5,
0);

foreach($listeArticles as $article)
{
    // $article est une instance de Article
    echo $article->getContenu();
}
```

Cet exemple va récupérer toutes les entités ayant comme auteur « winzou » en les classant par date décroissante et en sélectionnant cinq (5) à partir du début (0). Elle retourne un Array également. Vous pouvez mettre plusieurs entrées dans le tableau des critères, afin d'appliquer plusieurs filtres.

findOneBy()

La méthode `findOneBy(array $ criteres)` fonctionne sur le même principe que la méthode `findBy()`, sauf qu'elle ne retourne qu'une seule entité. Les arguments `orderBy`, `limit` et `offset` n'existent donc pas. Exemple :

Code : PHP

```
<?php
getRepository = $this->getDoctrine()
->getManager()
->getRepository('SdzBlogBundle:Article');

$article = $repository->findOneBy(array('titre' => 'Mon dernier
weekend'));
// $article est une instance de Article
```

Ces méthodes permettent de couvrir pas mal de besoins. Mais pour aller plus loin encore, Doctrine nous offre deux autres méthodes magiques.

Les méthodes magiques

Vous connaissez le principe des [méthodes magiques](#), comme `__call()` qui émule des méthodes. Ces méthodes émulées n'existent pas dans la classe, elle sont prises en charge par `__call()` qui va exécuter du code en fonction du nom de la méthode appelée.

Voici les deux méthodes gérées par `__call()` dans les repositories.

`findByX($valeur)`

Première méthode, en remplaçant « X » par le nom d'une propriété de votre entité. Dans notre cas, pour l'entité Article, nous avons donc plusieurs méthodes : `findByTitre()`, `findByDate()`, `findByAuteur()`, `findByContenu()`, etc.

Cette méthode fonctionne comme `findBy()`, sauf que vous ne pouvez mettre qu'un seul critère, celui du nom de la méthode.

Attention, la limite de cette méthode est que vous ne pouvez pas utiliser les arguments pour trier, ni pour mettre une limite.

Code : PHP

```
<?php
$repository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');

$listeArticles = $repository->findByAuteur('winzou');
// $listeArticles est un Array qui contient tous les articles écrits
par winzou
```

`findOneByX($valeur)`

Deuxième méthode, en remplaçant « X » par le nom d'une propriété de votre entité. Dans notre cas, pour l'entité Article, nous avons donc plusieurs méthodes : `findOneByTitre()`, `findOneByDate()`, `findOneByAuteur()`, `findOneByContenu()`, etc.

Cette méthode fonctionne comme `findOneBy()`, sauf que vous ne pouvez mettre qu'un seul critère, celui du nom de la méthode.

Code : PHP

```
<?php
$repository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');

$article = $repository->findOneByTitre('Mon dernier weekend');
// $article est une instance d'Article
```

Toutes ces méthodes permettent de récupérer vos entités dans la plupart des cas. Simplement, elles montrent rapidement leurs limites lorsqu'on doit faire des jointures, ou effectuer des conditions plus complexes. Pour cela — et cela nous arrivera très souvent — il faudra faire nos propres méthodes de récupération.

Les méthodes de récupération personnelles

La théorie

Pour effectuer nos propres méthodes, il faut bien comprendre comment fonctionne Doctrine2 pour effectuer ses requêtes. Il faut notamment distinguer trois types d'objets qui vont nous servir, et qu'il ne faut pas confondre : le QueryBuilder, la Query et les résultats.

Le QueryBuilder

On l'a déjà vu rapidement, le QueryBuilder permet de construire une Query, mais il n'est pas une Query !

Pour récupérer un QueryBuilder, on peut utiliser simplement l'EntityManager. En effet, il dispose d'une méthode `createQueryBuilder()` qui nous retournera une instance de QueryBuilder. L'EntityManager est accessible depuis un repository en utilisant l'attribut `_em` d'un repository, soit `<?php $this->_em`. Le code complet pour récupérer un QueryBuilder neuf depuis une méthode d'un repository est donc `<?php $this->_em->createQueryBuilder()`.

Cependant, cette méthode nous retourne un QueryBuilder vide, c'est-à-dire sans rien de prédefini. C'est dommage, car lorsqu'on récupère un QueryBuilder depuis un repository, c'est que l'on veut faire une requête sur l'entité gérée par ce repository. Donc si l'on pouvait définir la partie `SELECT article FROM SdzBlogBundle:Article` sans trop d'effort, cela serait bien pratique, car ce qui est intéressant, c'est le reste de la requête. Heureusement, le repository contient également une méthode `createQueryBuilder($alias)` qui utilise la méthode de l'EntityManager, mais en définissant pour nous le SELECT et le FROM. Vous pouvez jeter un œil à [cette méthode `createQueryBuilder\(\)`](#) pour comprendre.

L'alias en argument de la méthode est le raccourci que l'on donne à l'entité du repository. On utilise souvent la première lettre du nom de l'entité, dans notre exemple de l'article cela serait donc un « a ».

Beaucoup de théorie, passons donc à la pratique ! Pour bien comprendre la différence QueryBuilder/Query, ainsi que la récupération du QueryBuilder, rien de mieux qu'un exemple. Nous allons recréer la méthode `findAll()` dans notre repository Article :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\EntityRepository;

/**
 * ArticleRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class ArticleRepository extends EntityRepository
{
    public function myfindAll()
    {
        $queryBuilder = $this->createQueryBuilder('a');

        // Méthode équivalente, mais plus longue :
        $queryBuilder = $this->_em->createQueryBuilder()
            ->select('a')
            ->from($this->_entityName, 'a');
        // Dans un repository, $this->_entityName est le namespace de
        l'entité gérée
        // Ici, il vaut donc Sdz\BlogBundle\Entity\Article

        // On a fini de construire notre requête

        // On récupère la Query à partir du QueryBuilder
        $query = $queryBuilder->getQuery();

        // On récupère les résultats à partir de la Query
        $resultats = $query->getResult();

        // On retourne ces résultats
        return $resultats;
    }
}

```

Cette méthode `myFindAll()` retourne exactement le même résultat qu'un `findAll()`, c'est-à-dire un tableau de toutes les entités `Article` dans notre base de données. Vous pouvez le voir, faire une simple requête est très facile. Pour mieux le visualiser, je vous propose la même méthode sans les commentaires et en raccourci :

Code : PHP

```
<?php
public function myFindAll()
{
    return $this->createQueryBuilder('a')
        ->getQuery()
        ->getResult();
}
```

Simplissime, non ? 

Et bien sûr, pour récupérer les résultats depuis un contrôleur, le code est le suivant :

Code : PHP

```
<?php
public function testAction()
{
    $liste_articles = $this->getDoctrine()
        ->getManager()
        ->getRepository('SdzBlogBundle:Article')
        ->myFindAll();

    // Reste de la méthode du contrôleur
}
```

Sauf que pour l'instant on a juste récupéré le `QueryBuilder`, on n'a pas encore joué avec lui. Il dispose de plusieurs méthodes afin de construire notre requête. Il y a une ou plusieurs méthodes par partie de requête : le WHERE, le ORDER BY, le FROM, etc. Elles n'ont rien de compliqué, voyez-le dans les exemples suivants.

Commençons par une méthode équivalente au `find($id)` de base, pour nous permettre de manipuler le `where()` et le `setParameter()`.

Code : PHP

```
<?php
// Dans un repository

public function myFindOne($id)
{
    // On passe par le QueryBuilder vide de l'EntityManager pour
    // l'exemple
    $qb = $this->_em->createQueryBuilder();

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

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

Vous connaissez le principe des paramètres, qui est le même qu'avec PDO. On définit un paramètre dans la requête avec

:nom_du_parametre, puis on attribue une valeur à ce paramètre avec la méthode setParameter('nom_du_parametre', \$valeur).

Voici un autre exemple pour utiliser le andWhere() ainsi que le orderBy(). Créons une méthode pour récupérer tous les articles écrits par un auteur avant une année donnée :

Code : PHP

```
<?php
// Depuis un repository

public function findByAuteurAndDate($auteur, $annee)
{
    // On utilise le QueryBuilder créé par le repository directement
    // pour gagner du temps
    // Plus besoin de faire le select() ni le from() par la suite
    // donc
    $qb = $this->createQueryBuilder('a');

    $qb->where('a.auteur = :auteur')
        ->setParameter('auteur', $auteur)
        ->andWhere('a.date < :annee')
        ->setParameter('annee', $annee)
        ->orderBy('a.date', 'DESC');

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

Maintenant, voyons un des avantages du QueryBuilder. Vous vous en souvenez, je vous avais parlé d'une méthode pour centraliser une condition par exemple. Voyons donc une application de ce principe, en considérant que la condition « articles postés durant l'année en cours » est une condition dont on va se resservir souvent. Il faut donc en faire une méthode, que voici :

Code : PHP

```
<?php
// Depuis un repository

public function whereCurrentYear(\Doctrine\ORM\QueryBuilder $qb)
{
    $qb->andWhere('a.date BETWEEN :debut AND :fin')
        ->setParameter('debut', new \Datetime(date('Y') . '-01-01')) // Date entre le 1er janvier de cette année
        ->setParameter('fin', new \Datetime(date('Y') . '-12-31')); // Et le 31 décembre de cette année

    return $qb;
}
```

Vous notez donc que cette méthode ne traite pas une Query, mais bien uniquement le QueryBuilder. C'est en cela que ce dernier est très pratique, car faire cette méthode sur une requête en texte simple est possible, mais compliqué. Il aurait fallu voir si le WHERE était déjà présent dans la requête, si oui mettre un AND au bon endroit, etc. Bref, pas simple.

Pour utiliser cette méthode, voici la démarche :

Code : PHP

```
<?php
// Depuis un repository

public function myFind()
```

```

{
    $qb = $this->createQueryBuilder('a');

    // On peut ajouter ce qu'on veut avant
    $qb->where('a.auteur = :auteur')
        ->setParameter('auteur', 'winzou');

    // On applique notre condition
    $qb = $this->whereCurrentYear($qb);

    // On peut ajouter ce qu'on veut après
    $qb->orderBy('a.date', 'DESC');

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

```

Voilà, vous pouvez dorénavant appliquer cette condition à n'importe laquelle de vos requêtes en construction.

Je ne vous ai pas listé toutes les méthodes du QueryBuilder, il en existe bien d'autres. Pour cela, vous devez absolument mettre la page suivante dans vos favoris : [http://www.doctrine-project.org/docs/0 \[...\] -builder.html](http://www.doctrine-project.org/docs/0 [...] -builder.html). Ouvrez-la et gardez-la sous la main à chaque fois que vous voulez faire une requête à l'aide du QueryBuilder, c'est la référence !

La Query

Vous l'avez vu, la Query est l'objet à partir duquel on extrait les résultats. Il n'y a pas grand-chose à savoir sur cet objet en lui-même, car il ne permet pas grand-chose à part récupérer les résultats. Il sert en fait surtout à la gestion du cache des requêtes. Un prochain chapitre est à venir sur ce cache de requêtes.

Mais détaillons tout de même les différentes façons d'extraire les résultats de la requête. Ces différentes manières sont toutes à maîtriser, car elles concernent chacune un type de requête.

`getResult()`

Exécute la requête et retourne un tableau contenant les résultats sous forme d'objets. Vous récupérez ainsi une liste des objets, sur lesquels vous pouvez faire des opérations, des modifications, etc.

Même si la requête ne retourne qu'un seul résultat, cette méthode retourne un tableau.

Code : PHP

```

<?php
$entites = $qb->getQuery()->getResult();

foreach($entites as $entite)
{
    // $entite est une instance d'Article pour notre exemple
    $entite->getAttribut();
}

```

`getArrayResult()`

Exécute la requête et retourne un tableau contenant les résultats sous forme de tableaux. Comme avec `getResult()`, vous récupérez un tableau même s'il n'y a qu'un seul résultat. Mais dans ce tableau, vous n'avez pas vos objets d'origine, vous avez des simples tableaux. Cette méthode est utilisée lorsque vous ne voulez que lire vos résultats, sans y apporter de modification. Elle est dans ce cas plus rapide que son homologue `getResult()`.

Code : PHP

```
<?php
$entites = $qb->getQuery()->getArrayResult();

foreach($entites as $entite)
{
    // $entite est un tableau
    // Faire $entite->getAttribut() est impossible. Vous devez faire :
    $entite['attribut'];
}
```

Heureusement, Twig est intelligent : {{ entite.attribut }} exécute \$entite->getAttribut() si \$entite est un objet, et exécute \$entite['attribut'] sinon. Du point de vue de Twig, vous pouvez utiliser getResult() ou getArrayResult() indifféremment.

getScalarResult()

Exécute la requête et retourne un tableau contenant les résultats sous forme de valeurs. Comme avec getResult(), vous récupérez un tableau même s'il n'y a qu'un seul résultat.

Mais dans ce tableau, un résultat est une valeur, non un tableau de valeurs (getArrayResult) ou un objet de valeurs (getResult). Cette méthode est donc utilisée lorsque vous ne sélectionnez qu'une seule valeur dans la requête, par exemple : **SELECT COUNT(*) FROM ...** Ici, la valeur est la valeur du COUNT.

Code : PHP

```
<?php
$entites = $qb->getQuery()->getScalarResult();

foreach($entites as $valeur)
{
    // $valeur est la valeur de ce qui a été sélectionné : un nombre,
    // un texte, etc.
    $valeur;

    // Faire $valeur->getAttribut() ou $valeur['attribut'] est
    // impossible
}
```

getOneOrNullResult()

Exécute la requête et retourne un seul résultat, ou null si pas de résultat. Cette méthode retourne donc une instance de l'entité (ou null) et non un tableau d'entités comme getResult().

Cette méthode déclenche une exception `Doctrine\ORM\NonUniqueResultException` si la requête retourne plus d'un seul résultat. Il faut donc l'utiliser si l'une de vos requêtes n'est pas censée retourner plus d'un résultat : déclencher une erreur plutôt que de laisser courir permet d'anticiper des futurs bugs !

Code : PHP

```
<?php
$entite = $qb->getQuery()->getOneOrNullResult();

// $entite est une instance d'Article dans notre exemple
// Ou null si la requête ne contient pas de résultat

// Et une exception a été déclenchée si plus d'un résultat
```

getSingleResult()

Exécute la requête et retourne un seul résultat. Cette méthode est exactement la même que `getOneOrNullResult()`, sauf qu'elle déclenche une exception `Doctrine\ORM\NoResultException` si aucun résultat.

C'est une méthode très utilisée, car faire des requêtes qui ne retournent qu'un unique résultat est très fréquent.

Code : PHP

```
<?php
$entite = $qb->getQuery()->getSingleResult();

// $entite est une instance d'Article dans notre exemple

// Une exception a été déclenchée si plus d'un résultat
// Une exception a été déclenchée si pas de résultat
```

getSingleScalarResult()

Exécute la requête et retourne une seule valeur, et déclenche des exceptions si pas de résultat ou plus d'un résultat.

Cette méthode est très utilisée également pour des requêtes du type `SELECT COUNT(*) FROM Article`, qui ne retournent qu'une seule ligne de résultat, et une seule valeur dans cette ligne.

Code : PHP

```
<?php
$valeur = $qb->getQuery()->getSingleScalarResult();

// $valeur est directement la valeur du COUNT dans la requête
// exemple

// Une exception a été déclenchée si plus d'un résultat
// Une exception a été déclenchée si pas de résultat
```

execute()

Exécute la requête. Cette méthode est utilisée principalement pour exécuter des requêtes qui ne retournent pas de résultats (des `UPDATE`, `INSERT INTO`, etc.).

Cependant, toutes les autres méthodes que nous venons de voir ne sont en fait que des raccourcis vers cette méthode `execute()`, en changeant juste le mode d'hydratation des résultats (objet, tableau, etc.).

Code : PHP

```
<?php
// Exécute un UPDATE par exemple :
$query->getQuery()->execute();

// Voici deux méthodes strictement équivalentes :
$resultats = $query->getArrayResult();
// Et :
$resultats = $query->execute(array(), Query::HYDRATE_ARRAY);

// Le premier argument de execute() est un tableau de paramètres
// Vous pouvez aussi passer par la méthode setParameter(), au choix
```

```
// Le deuxième argument de execute() est ladite méthode
d'hydratation
```

Pensez donc à bien choisir votre façon de récupérer les résultats à chacune de vos requêtes.

Utilisation du Doctrine Query Language (DQL)

Le DQL est une sorte de SQL adapté à l'ORM Doctrine2. Il permet de faire des requêtes un peu à l'ancienne, en écrivant une requête en chaîne de caractères (en opposition au QueryBuilder).

Pour écrire une requête en DQL, il faut donc oublier le QueryBuilder, on utilisera seulement l'objet Query. Et la méthode pour récupérer les résultats sera la même. Le DQL n'a rien de compliqué, et il est très bien documenté.

La théorie

Pour créer une requête en utilisant du DQL, il faut utiliser la méthode `createQuery()` de l'EntityManager :

Code : PHP

```
<?php
// Depuis un repository
public function myFindAllDQL()
{
    $query = $this->_em->createQuery('SELECT a FROM
SdzBlogBundle:Article a');
    $resultats = $query->getResult();

    return $resultats;
}
```

Regardons de plus près la requête DQL en elle-même :

Code : SQL

```
SELECT a FROM SdzBlogBundle:Article a
```

Tout d'abord, vous voyez que l'on n'utilise pas de table. On a dit qu'on pensait objet et non plus base de données ! Il faut donc utiliser dans les FROM et les JOIN le nom des entités. Soit en utilisant le nom raccourci comme on l'a fait, soit le namespace complet de l'entité. De plus, il faut toujours donner un alias à l'entité, ici on a mis « a ». On met souvent la première lettre de l'entité, même si ce n'est absolument pas obligatoire.

Ensuite, vous imaginez bien qu'il ne faut pas sélectionner un à un les attributs de nos entités, cela n'aurait pas de sens. Une entité Article avec le titre renseigné mais pas la date ? Ce n'est pas logique. C'est pourquoi on sélectionne simplement l'alias, ici « a », ce qui sélectionne en fait tous les attributs d'un article. L'équivalent d'une étoile (*) en SQL donc.

 Sachez qu'il est tout de même possible de ne sélectionner qu'une partie d'un objet, en faisant « a.titre » par exemple. Mais vous ne recevez alors qu'un tableau contenant les attributs sélectionnés, et non un objet. Vous ne pouvez donc pas modifier/supprimer/etc. l'objet, puisque c'est un tableau. Cela sert dans des requêtes particulières, mais la plupart du temps on sélectionnera bien tout l'objet.

Faire des requêtes en DQL n'a donc rien de compliqué. Lorsque vous les faites, gardez bien sous la main [la page de la documentation sur le DQL](#) pour en connaître la syntaxe. En attendant, je peux vous montrer quelques exemples afin que vous ayez une idée globale du DQL.

Pour tester rapidement vos requêtes DQL sans avoir à les implémenter dans une méthode de votre repository, Doctrine2 nous simplifie la vie grâce à la commande `doctrine:query:dql`. Cela vous permet de faire quelques tests afin de construire ou de vérifier vos requêtes, à utiliser sans modération donc ! Je vous invite dès maintenant à exécuter la commande suivante :

```
php app/console doctrine:query:dql "SELECT a FROM SdzBlogBundle:Article a".
```

Exemples

Pour faire une jointure :

Code : SQL

```
SELECT a, u FROM Article a JOIN a.utilisateur u WHERE u.age = 25
```

Pour utiliser une fonction SQL :

Code : SQL

```
SELECT a FROM Article a WHERE TRIM(a.auteur) = 'winzou'
```

Pour sélectionner seulement un attribut (attention les résultats seront donc sous forme de tableaux et non d'objets) :

Code : SQL

```
SELECT a.titre FROM Article a WHERE a.id IN(1, 3, 5)
```

Et bien sûr vous pouvez également utiliser des paramètres :

Code : PHP

```
<?php
public function myFindDQL($id)
{
    $query = $this->_em->createQuery('SELECT a FROM Article a WHERE
a.id = :id');
    $query->setParameter('id', $id);
    return $query->getSingleResult(); // Utilisation de
getSingleResult car la requête ne doit retourner qu'un seul
résultat
}
```

Utiliser les jointures dans nos requêtes

Pourquoi utiliser les jointures ?

Je vous en ai déjà parlé dans le chapitre précédent sur les relations entre entités. Lorsque vous utilisez la syntaxe `$entiteA->getEntiteB()`, Doctrine exécute une requête afin de charger les entités B qui sont liées à l'entité A.

L'objectif est donc d'avoir la maîtrise sur quand charger juste l'entité A, et quand charger l'entité A avec ses entités B liées. Nous avons déjà vu le premier cas, par exemple un `$repositoryA->find($id)` ne récupère qu'une seule entité A sans récupérer les entités liées. Maintenant, voyons comment réaliser le deuxième cas, c'est-à-dire récupérer tout d'un coup avec une jointure, pour éviter une seconde requête par la suite.

Tout d'abord, rappelons le cas d'utilisation principal de ces jointures. C'est surtout lorsque vous bouclez sur une liste d'entités A

(par exemple des articles), et que dans cette boucle vous faites `$entiteA->getEntiteB()` (par exemple des commentaires). Avec une requête par itération dans la boucle, vous explosez votre nombre de requêtes sur une seule page ! C'est donc principalement pour éviter cela que nous allons faire des jointures.

Comment faire des jointures avec le QueryBuilder ?

Heureusement, c'est très simple ! Voici tout de suite un exemple :

Code : PHP

```
<?php
// Depuis le repository d'Article
public function getArticleAvecCommentaires()
{
    $qb = $this->createQueryBuilder('a')
        ->leftJoin('a.commentaires', 'c')
        ->addSelect('c');

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

L'idée est donc très simple :

- D'abord on crée une jointure avec la méthode `leftJoin()` (ou `join()` pour faire l'équivalent d'un `INNER JOIN`). Le premier argument de la méthode est l'attribut de l'entité principale (celle qui est dans le `FROM` de la requête) sur lequel faire la jointure. Dans l'exemple, l'entité `Article` possède un attribut `commentaires`. Le deuxième argument de la méthode est l'alias de l'entité jointe.
- Puis on sélectionne également l'entité jointe, via un `addSelect()`. En effet, un `select()` tout court aurait écrasé le `select('a')` déjà fait par le `createQueryBuilder()`, rappelez-vous.

Attention : on ne peut faire une jointure que si l'entité du `FROM` possède un attribut vers l'entité à joindre ! Cela veut dire que soit l'entité du `FROM` est l'entité propriétaire de la relation, soit la relation est bidirectionnelle.

Dans notre exemple, la relation entre `Article` et `Commentaire` est une *Many-To-One* avec `Commentaire` le côté `Many`, le côté propriétaire donc. Cela veut dire que pour pouvoir faire la jointure dans ce sens, la relation est bidirectionnelle, afin d'ajouter un attribut `commentaires` dans l'entité inverse `Article`.



Et pourquoi n'a-t-on pas précisé la condition « ON » du JOIN ?

C'est une bonne question. La réponse est très logique, pour cela réfléchissez plutôt à la question suivante : pourquoi est-ce qu'on rajoute un `ON` habituellement dans nos requêtes SQL ? C'est pour que MySQL (ou tout autre SGBDR) puisse savoir sur quelle condition faire la jointure. Or ici, on s'adresse à Doctrine et non directement à MySQL. Et bien entendu, Doctrine connaît déjà tout sur notre association, grâce aux annotations !

Bien sûr, vous pouvez toujours personnaliser la condition de jointure, en rajoutant vos conditions à la suite du `ON` généré par Doctrine, grâce à la syntaxe du `WITH` :

Code : PHP

```
<?php
$qb->join('a.commentaires', 'c', 'WITH', 'YEAR(c.date) > 2011')
```

Le troisième argument est le type de condition `WITH`, et le quatrième argument est ladite condition.



« WITH » ? C'est quoi cette syntaxe pour faire une jointure ?

En SQL, la différence entre le **ON** et le **WITH** est simple : un **ON** définit la condition pour la jointure, alors qu'un **WITH** ajoute une condition pour la jointure. Attention, en DQL le **ON** n'existe pas, seul le **WITH** est supporté. Ainsi, la syntaxe précédente avec le **WITH** serait équivalente à la syntaxe SQL suivante à base de **ON** :

Code : SQL

```
SELECT * FROM Article a JOIN Commentaire c ON c.article = a.id AND
YEAR(c.date) > 2011
```

Grâce au **WITH**, on n'a pas besoin de réécrire la condition par défaut de la jointure, le **c.article = a.id**.

Comment utiliser les jointures ?

Réponse : comme d'habitude ! Vous n'avez rien à modifier dans votre code. Si vous utilisez une entité dont vous avez récupéré les entités liées avec une jointure, vous pouvez alors utiliser les getters joyeusement sans craindre de requête supplémentaire. Reprenons l'exemple de la méthode `getArticleAvecCommentaires()` définie précédemment, on pourrait utiliser les résultats comme ceci :

Code : PHP

```
<?php
// Depuis un contrôleur
public function listeAction()
{
    $listeArticles = $this->getDoctrine()
        ->getManager()
        ->getRepository('SdzBlogBundle:Article')
        ->getArticleAvecCommentaires();

    foreach($listeArticles as $article)
    {
        // Ne déclenche pas de requête : les commentaires sont déjà
        chargés !
        // Vous pourriez faire une boucle dessus pour les afficher tous
        $article->getCommentaires();
    }

    // ...
}
```

Voici donc comment vous devrez faire la plupart de vos requêtes. En effet, vous aurez souvent besoin d'utiliser des entités liées entre elles, et faire une ou plusieurs jointures s'impose très souvent. 😊

Application : les entités de notre blog

Plan d'attaque

Nous allons ajouter une méthode dans l'`ArticleRepository` pour récupérer tous les articles qui correspondent à une liste de catégories. La définition de la méthode est donc `<?php getAvecCategories(array $nom_categories) ?>`, que l'on pourra utiliser comme exemple : `<?php $articleRepository->getAvecCategories(array('Doctrine2', 'Tutoriel')) ?>`.

À vous de jouer !



Important : **faites-le vous-mêmes** ! La correction est juste en dessous, je sais, mais si vous ne faites pas *maintenant* l'effort d'y réfléchir par vous-mêmes, cela vous handicadera par la suite !

Le code

ArticleRepository.php :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\EntityRepository;

/**
 * ArticleRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class ArticleRepository extends EntityRepository
{
    public function getAvecCategories(array $nom_categories)
    {
        $qb = $this->createQueryBuilder('a');

        // On fait une jointure avec l'entité Categorie, avec pour
        alias « c »
        $qb ->join('a.categories', 'c')
            ->where($qb->expr()->in('c.nom', $nom_categories)); // Puis
        on filtre sur le nom des catégories à l'aide d'un IN

        // Enfin, on retourne le résultat
        return $qb->getQuery()
            ->getResult();
    }
}
```



Que faire avec ce que retourne cette fonction ?

Comme je l'ai dit précédemment, cette fonction va retourner un tableau d'Article. Qu'est-ce que l'on veut en faire ? Les afficher. Donc la première chose à faire est de passer ce tableau à Twig. Ensuite, dans Twig, vous faites un simple { % for %} pour afficher les articles. Ce n'est vraiment pas compliqué à utiliser !

Et voilà, vous avez tout le code. Je n'ai qu'une chose à vous dire à ce stade du cours : entraînez-vous ! Amusez-vous à faire des requêtes dans tous les sens dans l'ArticleRepository ou même dans les autres repositories. Jouez avec les relations entre les entités, créez-en d'autres. Bref, cela ne viendra pas tout seul, il va falloir travailler un peu de votre côté. 😊

En résumé

- Le rôle d'un repository est, à l'aide du langage DQL ou du constructeur de requêtes, de récupérer des entités selon des contraintes, des tris, etc.
- Un repository dispose toujours de quelques méthodes de base, permettant de récupérer de façon très simple les entités.
- Mais la plupart du temps, il faut créer des méthodes personnelles pour récupérer les entités exactement comme on le veut.
- Il est indispensable de faire les bonnes jointures afin de limiter au maximum le nombre de requêtes SQL sur vos pages.

Les évènements et extensions Doctrine

Maintenant que vous savez manipuler vos entités, vous allez vous rendre compte que pas mal de comportements sont répétitifs. En bon développeurs, il est hors de question de dupliquer du code ou de perdre du temps : nous sommes bien trop fainéants !

Ce chapitre a pour objectif de vous présenter les évènements et les extensions Doctrine, qui vous permettront de simplifier certains cas usuels que vous rencontrerez.

Les évènements Doctrine

L'intérêt des évènements Doctrine

Dans certains cas, vous pouvez avoir besoin d'effectuer des actions juste avant ou juste après la création, la mise à jour ou la suppression d'une entité. Par exemple, si vous stockez la date d'édition d'un article, à chaque modification de l'entité Article il faut mettre à jour cet attribut juste avant la mise à jour dans la base de données.

Ces actions, vous devez les faire à *chaque fois*. Cet aspect systématique a deux impacts. D'une part, cela veut dire qu'il faut être sûrs de vraiment les effectuer à chaque fois pour que votre base de données soit cohérente. D'autre part, cela veut dire qu'on est bien trop fainéants pour se répéter !

C'est ici qu'interviennent les évènements Doctrine. Plus précisément, vous les trouverez sous le nom de *callbacks* du cycle de vie (*lifecycle* en anglais) d'une entité. Un *callback* est une méthode de votre entité, et on va dire à Doctrine de l'exécuter à certains moments.

On parle d'évènements de « cycle de vie », car ce sont différents évènements que Doctrine lève à chaque moment de la vie d'une entité : son chargement depuis la base de données, sa modification, sa suppression, etc. On en reparle plus loin, je vous dresserai une liste complète des évènements et de leur utilisation.

Définir des callbacks de cycle de vie

Pour vous expliquer le principe, nous allons prendre l'exemple de notre entité Article, qui va comprendre un attribut \$dateEdition représentant la date de la dernière édition de l'article. Si vous ne l'avez pas déjà, ajoutez-le maintenant, et n'oubliez pas de mettre à jour la base de données à l'aide de la commande doctrine:schema:update .

1. Définir l'entité comme contenant des callbacks

Tout d'abord, on doit dire à Doctrine que notre entité contient des callbacks de cycle de vie ; cela se définit grâce à l'annotation HasLifecycleCallbacks dans le namespace habituel des annotations Doctrine :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Article
{
    // ...
}
```

Cette annotation permet à Doctrine de vérifier les callbacks éventuels contenus dans l'entité. Ne l'oubliez pas, car sinon vos différents callbacks seront tout simplement ignorés.

2. Définir un callback et ses événements associés

Maintenant, il faut définir des méthodes et surtout, les évènements sur lesquels elles seront exécutées.

Continuons dans notre exemple, et créons une méthode `updateDate()` dans l'entité `Article`. Cette méthode doit définir l'attribut `$dateEdition` à la date actuelle, afin de mettre à jour automatiquement la date d'édition d'un article. Voici à quoi elle pourrait ressembler :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Article
{
    // ...

    public function updateDate()
    {
        $this->setDateEdition(new \Datetime());
    }
}
```

Maintenant il faut dire à Doctrine d'exécuter cette méthode (ce *callback*) dès que l'entité `Article` est modifiée. On parle d'*écouter un évènement*. Il existe plusieurs évènements de cycle de vie avec Doctrine, celui qui nous intéresse ici est l'évènement `PreUpdate` : c'est-à-dire que la méthode va être exécutée juste avant que l'entité ne soit modifiée en base de données. Voici à quoi cela ressemble :

Code : PHP

```
<?php

/**
 * @ORM\PreUpdate
 */
public function updateDate()
```

C'est tout !

Vous pouvez dès à présent tester le comportement. Essayez de faire un petit code de test pour charger un article, le modifier, et l'enregistrer (avec un `flush()`), vous verrez que l'attribut `$dateEdition` va se mettre à jour automatiquement. Attention, l'évènement `update` n'est pas déclenché à la création d'une entité, mais seulement à sa modification : c'est parfaitement ce qu'on veut dans notre exemple.

Pour aller plus loin, il y a deux points qu'il vous faut savoir. D'une part, au même titre que l'évènement `PreUpdate`, il existe l'évènement `postUpdate` et bien d'autres, on en dresse une liste dans le tableau suivant. D'autre part, vous l'avez sûrement noté, mais le *callback* ne prend aucun argument, vous ne pouvez en effet utiliser et modifier que l'entité courante. Pour exécuter des actions plus complexes lors d'évènements, il faut créer des services, on voit cela plus loin.

Liste des évènements de cycle de vie

Les différents évènements du cycle de vie sont récapitulés dans le tableau suivant.

Méta-événement	Évènement	Description
Persist	PrePersist	L'évènement <code>prePersist</code> se produit juste avant que l'EntityManager ne persiste effectivement l'entité. Concrètement, cela exécute le <code>callback</code> juste avant un <code>\$em->persist(\$entity)</code> . Il ne concerne que les entités nouvellement créées. Du coup, il y a deux conséquences : d'une part, les modifications que vous apportez à l'entité seront persistées en base de données, puisqu'elles sont effectives avant que l'EntityManager n'enregistre l'entité en base. D'autre part, vous n'avez pas accès à l'id de l'entité si celui-ci est autogénéré, car justement l'entité n'est pas encore enregistrée en base de données, et donc l'id pas encore généré.
	PostPersist	L'évènement <code>postPersist</code> se produit juste après que l'EntityManager a effectivement persisté l'entité. Attention, cela n'exécute pas le <code>callback</code> juste après le <code>\$em->persist(\$entity)</code> , mais juste après le <code>\$em->flush()</code> . À l'inverse du <code>prePersist</code> , les modifications que vous apportez à l'entité ne seront pas persistées en base (mais seront tout de même appliquées à l'entité, attention) ; mais vous avez par contre accès à l'id qui a été généré lors du <code>flush()</code> .
Update	PreUpdate	L'évènement <code>preUpdate</code> se produit juste avant que l'EntityManager ne modifie une entité. Par modifiée, j'entends que l'entité existait déjà, que vous y avez apporté des modifications, puis un <code>\$em->flush()</code> . Le <code>callback</code> sera exécuté juste avant le <code>flush()</code> . Attention, il faut que vous ayez modifié au moins un attribut pour que l'EntityManager génère une requête et donc déclenche cet évènement. Vous avez accès à l'id autogénéré (car l'entité existe déjà), et vos modifications seront persistées en base de données.
	PostUpdate	L'évènement <code>postUpdate</code> se produit juste après que l'EntityManager a effectivement modifié une entité. Vous avez accès à l'id et vos modifications ne sont pas persistées en base de données.
Remove	PreRemove	L'évènement <code>preRemove</code> se produit juste avant que l'EntityManager ne supprime une entité, c'est-à-dire juste avant un <code>\$em->flush()</code> qui précède un <code>\$em->remove(\$entity)</code> . Attention, soyez prudents dans cet évènement, si vous souhaitez supprimer des fichiers liés à l'entité par exemple, car à ce moment l'entité n'est pas encore effectivement supprimée, et la suppression peut être annulée en cas d'erreur dans une des opérations à effectuer dans le <code>flush()</code> .
	PostRemove	L'évènement <code>postRemove</code> se produit juste après que l'EntityManager a effectivement supprimé une entité. Si vous n'avez plus accès à son id, c'est ici que vous pouvez effectuer une suppression de fichier associé par exemple.
Load	PostLoad	L'évènement <code>postLoad</code> se produit juste après que l'EntityManager a chargé une entité (ou après un <code>\$em->refresh()</code>). Utile pour appliquer une action lors du chargement d'une entité.



Attention, ces évènements se produisent lorsque vous créez et modifiez vos entités en manipulant les objets. Ils ne sont pas déclenchés lorsque vous effectuez des requêtes DQL ou avec le `QueryBuilder`. Car ces requêtes peuvent toucher un grand nombre d'entités et il serait dangereux pour Doctrine de déclencher les évènements correspondants un à un.

Un autre exemple d'utilisation

Pour bien comprendre l'intérêt des évènements, je vous propose un deuxième exemple : un compteur de commentaires pour les articles du blog.

L'idée est la suivante : nous avons un blog très fréquenté, et un petit serveur. Au lieu de récupérer le nombre de commentaires des articles à l'aide d'une requête `COUNT(*)`, on décide de rajouter un attribut `nbCommentaires` à notre entité `Article`. L'enjeu maintenant est de tenir cet attribut parfaitement à jour, et surtout très facilement.

C'est là que les évènements interviennent. Si on réfléchit un peu, le processus est assez simple et systématique :

- À chaque création d'un commentaire, on doit effectuer un +1 au compteur de l'`Article` lié ;
- À chaque suppression d'un commentaire, on doit effectuer un -1 au compteur de l'`Article` lié.

Ce genre de comportement, relativement simple et systématique, est typiquement ce que nous pouvons automatiser grâce aux événements Doctrine.

Les deux événements qui nous intéressent ici sont donc la création et la suppression d'un commentaire. Il s'agit des événements PrePersist et PreRemove. Pourquoi ? Car les événements *Update sont déclenchés à la mise à jour d'un Commentaire, ce qui ne change pas notre compteur ici. Et les événements Post* sont déclenchés après la mise à jour effective de l'entité dans la base de données, du coup la mise à jour de notre compteur ne serait pas enregistrée.

Au final, voici ce que nous devons rajouter dans l'entité Commentaire :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CommentaireRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Commentaire
{
    /**
     * @ORM\prePersist
     */
    public function increase()
    {
        $nbCommentaires = $this->getArticle()->getNbCommentaires();
        $this->getArticle()->setNbCommentaires($nbCommentaires+1);
    }

    /**
     * @ORM\preRemove
     */
    public function decrease()
    {
        $nbCommentaires = $this->getArticle()->getNbCommentaires();
        $this->getArticle()->setNbCommentaires($nbCommentaires-1);
    }

    // ...
}
```

Cette solution est possible car nous avons une relation entre ces deux entités Commentaire et Article, il est donc possible d'accéder à l'article depuis un commentaire.



Bien entendu, il vous faut rajouter l'attribut \$nbCommentaires dans l'entité Article si vous ne l'avez pas déjà.

Les extensions Doctrine

L'intérêt des extensions Doctrine

Dans la gestion des entités d'un projet, il y a des comportements assez communs que vous souhaiterez implémenter. Par exemple, il est très classique de vouloir générer des slugs pour les articles d'un blog, les sujets d'un forum, etc. Plutôt que de réinventer tout le comportement nous-mêmes, nous allons utiliser les extensions Doctrine !

Doctrine2 est en effet très flexible, et la communauté a déjà créé une série d'extensions Doctrine très pratiques afin de vous aider avec les tâches usuelles liées aux entités. À l'image des événements, utiliser ces extensions évite de se répéter au sein de votre application Symfony2 : c'est la philosophie DRY.

Installer le StofDoctrineExtensionBundle

Un bundle en particulier permet d'intégrer différentes extensions Doctrine dans un projet Symfony2, il s'agit de [StofDoctrineExtensionBundle](#). Commençons par l'installer avec Composer, rajoutez cette dépendance dans votre `composer.json` :

Code : JavaScript

```
// composer.json

"require": {
    "stof/doctrine-extensions-bundle": "dev-master"
}
```

Ce bundle intègre la bibliothèque [DoctrineExtensions](#) sous-jacente, qui est celle qui inclut réellement les extensions.

N'oubliez pas d'enregistrer le bundle dans le noyau :

Code : PHP

```
<?php
// app/AppKernel.php

public function registerBundles()
{
    return array(
        // ...
        new
Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        // ...
    );
}
```

Voilà le bundle est installé, il faut maintenant activer telle ou telle extension.

Utiliser une extension : l'exemple de Sluggable

L'utilisation des différentes extensions est très simple grâce à la flexibilité de Doctrine2 et au bundle pour Symfony2. Voici par exemple l'utilisation de l'extension Sluggable, qui permet de définir très facilement un attribut `slug` dans une entité : le `slug` sera automatiquement généré !

Tout d'abord, il faut activer l'extension Sluggable, il faut pour cela configurer le bundle via le fichier de configuration `config.yml`. Rajoutez donc cette section :

Code : YAML

```
# app/config/config.yml

# Stof\DoctrineExtensionBundle configuration
stof_doctrine_extensions:
    orm:
        default:
            sluggable: true
```

Cela va activer l'extension Sluggable. De la même manière, vous pourrez activer les autres extensions en les rajoutant à la suite.

Concrètement, l'utilisation des extensions se fait grâce à de judicieuses annotations. Vous l'aurez deviné, pour l'extension Sluggable, l'annotation est tout simplement `Slug`. En l'occurrence, il faut ajouter un nouvel attribut `slug` (le nom est arbitraire) dans votre entité, sur lequel nous mettrons l'annotation. Voici un exemple dans notre entité `Article` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Gedmo\Mapping\Annotation as Gedmo;

class Article
{
    // ...

    /**
     * @Gedmo\Slug(fields={"titre"})
     * @ORM\Column(length=128, unique=true)
     */
    private $slug;

    // ...
}
```

Dans un premier temps, vous avez l'habitude, on utilise le namespace de l'annotation, ici `Gedmo\Mapping\Annotation`.

Ensuite, l'annotation `Slug` s'utilise très simplement sur un attribut qui va contenir le slug. L'option `fields` permet de définir le ou les attributs à partir desquels le slug sera généré : ici le titre uniquement. Mais vous pouvez en indiquer plusieurs en les séparant par des virgules.



N'oubliez pas de mettre à jour votre base de données avec la commande `doctrine:schema:update`, mais également de générer le getter et le setter du `slug`, grâce à la commande `generate:doctrine:entities SdzBlogBundle:Article`.

C'est tout ! Vous pouvez dès à présent tester le nouveau comportement de votre entité. Créez une entité avec un titre de test, et enregistrez-la : son attribut `slug` sera automatiquement rempli ! Par exemple :

Code : PHP

```
<?php
// Dans un contrôleur

public function testAction()
{
    $article = new Article();
    $article->setTitre("L'histoire d'un bon weekend !");

    $em = $this->getDoctrine()->getManager();
    $em->persist($article);
    $em->flush(); // C'est à ce moment qu'est généré le slug

    return new Response('Slug généré : '.$article->getSlug()); // Affiche « Slug généré : l-histoire-d-un-bon-weekend »
}
```

L'attribut `slug` est rempli automatiquement par le bundle. Ce dernier utilise en réalité tout simplement les événements Doctrine

PrePersist et PreUpdate, qui permettent d'intervenir juste avant l'enregistrement et la modification de l'entité comme on l'a vu plus haut.

 Vous avez pu remarquer que j'ai défini l'attribut `slug` comme unique (`unique=true` dans l'annotation `Column`). En effet, dans le cadre d'un article de blog, on se sert souvent du slug comme identifiant de l'article, afin de l'utiliser dans les URL et améliorer le référencement. Sachez que l'extension est intelligente : si vous ajouter un Article avec un titre qui existe déjà, le slug sera suffixé de « -1 » pour garder l'unicité, par exemple « un-super-weekend-1 ». Si vous ajoutez un troisième titre identique, alors le slug sera « un-super-weekend-2 », etc.

Vous savez maintenant utiliser l'extension Doctrine Sluggable ! Voyons les autres extensions disponibles.

Liste des extensions Doctrine

Voici la liste de toutes les extensions actuellement disponibles, ainsi que leur description et des liens vers la documentation pour vous permettre de les implémenter dans votre projet.

Extension	Description
Tree	L'extension Tree automatise la gestion des arbres et ajoute des méthodes spécifiques au repository. Les arbres sont une représentation d'entités avec des liens type parents-enfants, utiles pour les catégories d'un forum par exemple.
Translatable	L'extension Translatable offre une solution aisée pour traduire des attributs spécifiques de vos entités dans différents langages. De plus, elle charge automatiquement les traductions pour la locale courante.
Sluggable	L'extension Sluggable permet de générer automatiquement un slug à partir d'attributs spécifiés.
Timestampable	L'extension Timestampable automatise la mise à jour d'attributs de type <code>date</code> dans vos entités. Vous pouvez définir la mise à jour d'un attribut à la création et/ou à la modification, ou même à la modification d'un attribut particulier. Vous l'aurez compris, cette extension fait la même chose que ce qu'on a fait dans le paragraphe précédent sur les événements Doctrine, et en mieux !
Loggable	L'extension Loggable permet de conserver les différentes versions de vos entités, et offre des outils de gestion des versions.
Sortable	L'extension Sortable permet de gérer des entités ordonnées, c'est-à-dire avec un ordre précis.
Softdeleteable	L'extension SoftDeleteable permet de « soft-supprimer » des entités, c'est-à-dire de ne pas les supprimer réellement, juste mettre un de leurs attributs à <code>true</code> pour les différencier. L'extension permet également de les filtrer lors des SELECT, pour ne pas utiliser des entités « soft-supprimées ».
Uploadable	L'extension Uploadable offre des outils pour gérer l'enregistrement de fichiers associés avec des entités. Elle inclut la gestion automatique des déplacements et des suppressions des fichiers.

Si vous n'avez pas besoin aujourd'hui de tous ces comportements, ayez-les en tête pour le jour où vous en trouverez l'utilité. Autant ne pas réinventer la roue si elle existe déjà ! 😊

Pour conclure

Ce chapitre touche à sa fin et marque la fin de la partie théorique sur Doctrine. Vous avez maintenant tous les outils pour gérer vos entités, et donc votre base de données. Surtout, n'hésitez pas à bien pratiquer, car c'est une partie qui implique de nombreuses notions : sans entraînement, pas de succès !

Le prochain chapitre est un TP permettant de mettre en pratique la plupart des notions abordées dans cette partie.

En résumé

- Les événements permettent de centraliser du code répétitif, afin de systématiser leur exécution et de réduire la duplication de code.
- Plusieurs événements jalonnent la vie d'une entité, afin de pouvoir exécuter une fonction aux endroits désirés.
- Les extensions permettent de reproduire des comportements communs dans une application, afin d'éviter de réinventer la

roue.

TP : Les entités de notre blog

L'objectif de ce chapitre est de mettre en application tout ce que nous avons vu au cours de cette partie sur Doctrine2. Nous allons créer les entités Article et Blog, mais également adapter le contrôleur pour nous en servir. Enfin, nous verrons quelques astuces de développement Symfony2 au cours du TP.

Surtout, je vous invite à bien essayer de réfléchir par vous-mêmes avant de lire les codes que je donne. C'est ce mécanisme de recherche qui va vous faire progresser sur Symfony2, il serait dommage de s'en passer !

Bon TP !

Synthèse des entités Entité Article

On a déjà pas mal traité l'entité Article au cours de cette partie. Pour l'instant, on a toujours le pseudo de l'auteur écrit en dur dans l'entité. Souvenez-vous, pour commencer, on n'a pas d'entité Utilisateur, on doit donc écrire le pseudo de l'auteur en dur dans les articles.

Voici donc la version finale de l'entité Article que vous devriez avoir :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Gedmo\Mapping\Annotation as Gedmo;

/**
 * Sdz\BlogBundle\Entity\Article
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Article
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

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

    /**
     * @var string $titre
     *
     * @ORM\Column(name="titre", type="string", length=255)
     */
    private $titre;

    /**
     * @var string $auteur
     *
     * @ORM\Column(name="auteur", type="string", length=255)
     */
    private $auteur;
}
```

```
/*
 * @ORM\Column(name="auteur")
 */
private $auteur;

/**
 * @ORM\Column(name="publication", type="boolean")
 */
private $publication;

/**
 * @var text $contenu
 *
 * @ORM\Column(name="contenu", type="text")
 */
private $contenu;

/**
 * @ORM\Column(type="date", nullable=true)
 */
private $dateEdition;

/**
 * @Gedmo\Slug(fields={"titre"})
 * @ORM\Column(length=128, unique=true)
 */
private $slug;

/**
 * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image",
 * cascade={"persist", "remove"})
 */
private $image;

/**
 * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Categorie",
 * cascade={"persist"})
 */
private $categories;

/**
 * @ORM\OneToMany(targetEntity="Sdz\BlogBundle\Entity\Commentaire",
 * mappedBy="article")
 */
private $commentaires; // Ici commentaires prend un « s », car un
article a plusieurs commentaires !

public function __construct()
{
    $this->date      = new \Datetime;
    $this->publication = true;
    $this->categories = new
\Doctrine\Common\Collections\ArrayCollection();
    $this->commentaires = new
\Doctrine\Common\Collections\ArrayCollection();
}

/**
 * @ORM\preUpdate
 * Callback pour mettre à jour la date d'édition à chaque
modification de l'entité
*/
public function updateDate()
{
    $this->setDateEdition(new \Datetime());
}

/**
 * @return integer
 */
public function getId()
```

```
    {
        return $this->id;
    }

    /**
 * @param datetime $date
 * @return Article
 */
    public function setDate(\Datetime $date)
    {
        $this->date = $date;
        return $this;
    }

    /**
 * @return datetime
 */
    public function getDate()
    {
        return $this->date;
    }

    /**
 * @param string $titre
 * @return Article
 */
    public function setTitre($titre)
    {
        $this->titre = $titre;
        return $this;
    }

    /**
 * @return string
 */
    public function getTitre()
    {
        return $this->titre;
    }

    /**
 * @param text $contenu
 * @return Article
 */
    public function setContenu($contenu)
    {
        $this->contenu = $contenu;
        return $this;
    }

    /**
 * @return text
 */
    public function getContenu()
    {
        return $this->contenu;
    }

    /**
 * @param boolean $publication
 * @return Article
 */
    public function setPublication($publication)
    {
        $this->publication = $publication;
        return $this;
    }

    /**
 * @return boolean
 */
```

```
/*
 * @param string $auteur
 * @return Article
 */
public function setAuteur($auteur)
{
    $this->auteur = $auteur;
    return $this;
}

/**
 * @return string
 */
public function getAuteur()
{
    return $this->auteur;
}

/**
 * @param Sdz\BlogBundle\Entity\Image $image
 * @return Article
 */
public function setImage(\Sdz\BlogBundle\Entity\Image $image = null)
{
    $this->image = $image;
    return $this;
}

/**
 * @return Sdz\BlogBundle\Entity\Image
 */
public function getImage()
{
    return $this->image;
}

/**
 * @param Sdz\BlogBundle\Entity\Categorie $categorie
 * @return Article
 */
public function addCategorie(\Sdz\BlogBundle\Entity\Categorie $categorie)
{
    $this->categories[] = $categorie;
    return $this;
}

/**
 * @param Sdz\BlogBundle\Entity\Categorie $categorie
 */
public function removeCategorie(\Sdz\BlogBundle\Entity\Categorie $categorie)
{
    $this->categories->removeElement($categorie);
}

/**
 * @return Doctrine\Common\Collections\Collection
 */
public function getCategories()
{
    return $this->categories;
}
```

```
 /**
 * @param Sdz\BlogBundle\Entity\Commentaire $commentaire
 * @return Article
 */
 public function addCommentaire(\Sdz\BlogBundle\Entity\Commentaire
$commentaire)
{
    $this->commentaires[] = $commentaire;
    return $this;
}

 /**
 * @param Sdz\BlogBundle\Entity\Commentaire $commentaire
 */
 public function removeCommentaire(\Sdz\BlogBundle\Entity\Commentaire
$commentaire)
{
    $this->commentaires->removeElement($commentaire);
}

 /**
 * @return Doctrine\Common\Collections\Collection
 */
 public function getCommentaires()
{
    return $this->commentaires;
}

 /**
 * @param datetime $dateEdition
 * @return Article
 */
 public function setDateEdition(\Datetime $dateEdition)
{
    $this->dateEdition = $dateEdition;
    return $this;
}

 /**
 * @return date
 */
 public function getDateEdition()
{
    return $this->dateEdition;
}

 /**
 * @param string $slug
 * @return Article
 */
 public function setSlug($slug)
{
    $this->slug = $slug;
    return $this;
}

 /**
 * @return string
 */
 public function getSlug()
{
    return $this->slug;
}
```

Entité Image

L'entité `Image` est une entité très simple, qui nous servira par la suite pour l'upload d'images avec Symfony2. Sa particularité est qu'elle peut être liée à n'importe quelle autre entité : elle n'est pas du tout exclusive à l'entité `Article`. Si vous souhaitez ajouter des images ailleurs que dans des `Article`, il n'y aura aucun problème.

Voici son code, que vous devriez déjà avoir :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Image.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ImageRepository")
 */
class Image
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $url
     *
     * @ORM\Column(name="url", type="string", length=255)
     */
    private $url;

    /**
     * @var string $alt
     *
     * @ORM\Column(name="alt", type="string", length=255)
     */
    private $alt;

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

    /**
     * @param string $url
     * @return Image
     */
    public function setUrl($url)
    {
        $this->url = $url;
        return $this;
    }

    /**
     * @return string
     */
    public function getUrl()
```

```

    {
        return $this->url;
    }

    /**
 * @param string $alt
 * @return Image
 */
    public function setAlt($alt)
{
    $this->alt = $alt;
    return $this;
}

/**
 * @return string
 */
    public function getAlt()
{
    return $this->alt;
}
}

```

Entité Commentaire

L'entité Commentaire, bien que très simple, contient la relation avec l'entité Article, c'est elle la propriétaire. Voici son code :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CommentaireRepository")
 */
class Commentaire
{
    /**
 * @ORM\Column(name="id", type="integer")
 * @ORM\Id
 * @ORM\GeneratedValue(strategy="AUTO")
 */
    private $id;

    /**
 * @ORM\Column(name="auteur", type="string", length=255)
 */
    private $auteur;

    /**
 * @ORM\Column(name="contenu", type="text")
 */
    private $contenu;

    /**
 * @ORM\Column(name="date", type="datetime")
 */
    private $date;
}

```

```
 /**
 * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article",
 * inverseBy="commentaires")
 * @ORM\JoinColumn(nullable=false)
 */
private $article;

public function __construct()
{
    $this->date = new \Datetime();
}

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

/**
 * @param string $auteur
 * @return Commentaire
 */
public function setAuteur($auteur)
{
    $this->auteur = $auteur;
    return $this;
}

/**
 * @return string
 */
public function getAuteur()
{
    return $this->auteur;
}

/**
 * @param text $contenu
 * @return Commentaire
 */
public function setContenu($contenu)
{
    $this->contenu = $contenu;
    return $this;
}

/**
 * @return text
 */
public function getContenu()
{
    return $this->contenu;
}

/**
 * @param datetime $date
 * @return Commentaire
 */
public function setDate(\Datetime $date)
{
    $this->date = $date;
    return $this;
}

/**
 * @return datetime
 */

```

```

    public function getDate()
    {
        return $this->date;
    }

    /**
 * @param Sdz\BlogBundle\Entity\Article $article
 * @return Commentaire
 */
    public function setArticle(\Sdz\BlogBundle\Entity\Article $article)
    {
        $this->article = $article;
        return $this;
    }

    /**
 * @return Sdz\BlogBundle\Entity\Article
 */
    public function getArticle()
    {
        return $this->article;
    }
}

```

Entité Categorie

L'entité Categorie ne contient qu'un attribut nom (enfin, vous pouvez en rajouter de votre côté bien sûr !). La relation avec Article est contenue dans l'entité Article, qui en est la propriétaire. Voici son code, que vous devriez déjà avoir :

Code : PHP

```

<?php
// src/Sdz/Bundle/Entity/Categorie.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CategorieRepository")
 */
class Categorie
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $nom
     *
     * @ORM\Column(name="nom", type="string", length=255)
     */
    private $nom;

    /**
     * @return integer
     */
}

```

```

public function getId()
{
    return $this->id;
}

< /**
 * @param string $nom
 * @return Categorie
 */
public function setNom($nom)
{
    $this->nom = $nom;
    return $this;
}

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

```

Entités Competence et ArticleCompetence

L'entité Competence ne contient, au même titre que l'entité Categorie, qu'un attribut nom, mais vous pouvez bien sûr en rajouter selon vos besoins. Voici son code :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Competence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

< /**
 * @ORM\Table()
 */
@ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CompetenceRepository")
class Competence
{
    < /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    < /**
     * @var string $nom
     *
     * @ORM\Column(name="nom", type="string", length=255)
     */
    private $nom;

    < /**
     * @return integer
     */

```

```

public function getId()
{
    return $this->id;
}

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

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

```

L'entité ArticleCompetence est l'entité de relation entre Article et Competence. Elle contient les attributs \$article et \$competence qui permettent de faire la relation, ainsi que d'autres attributs pour caractériser la relation, ici j'ai utilisé un attribut niveau. Voici son code, vous pouvez bien entendu rajouter les attributs de relation que vous souhaitez :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/ArticleCompetence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
* @ORM\Entity
*/
class ArticleCompetence
{
    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     */
    private $article;

    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Competence")
     */
    private $competence;

    /**
     * @ORM\Column()
     */
    private $niveau; // Ici j'ai un attribut de relation, que j'ai
    // appelé « niveau »

    /**
     * @param string $niveau
     * @return Article_Competence
     */
    public function setNiveau($niveau)
    {
        $this->niveau = $niveau;
        return $this;
    }
}

```

```

        $this->niveau = $niveau;
        return $this;
    }

    /**
 * @return string
 */
public function getNiveau()
{
    return $this->niveau;
}

/**
* @param Sdz\BlogBundle\Entity\Article $article
* @return ArticleCompetence
*/
public function setArticle(\Sdz\BlogBundle\Entity\Article
$article)
{
    $this->article = $article;
    return $this;
}

/**
* @return Sdz\BlogBundle\Entity\Article
*/
public function getArticle()
{
    return $this->article;
}

/**
* @param Sdz\BlogBundle\Entity\Competence $competence
* @return ArticleCompetence
*/
public function setCompetence(\Sdz\BlogBundle\Entity\Competence
$competence)
{
    $this->competence = $competence;
    return $this;
}

/**
* @return Sdz\BlogBundle\Entity\Competence
*/
public function getCompetence()
{
    return $this->competence;
}
}

```

Et bien sûr...

Si vous avez ajouté et/ou modifié des entités, n'oubliez pas de mettre à jour votre base de données ! Vérifiez les requêtes avec `php app/console doctrine:schema:update --dump-sql`, puis exécutez-les avec `--force`.

Adaptation du contrôleur

Théorie

Maintenant que l'on a nos entités, on va enfin pouvoir adapter notre contrôleur Blog pour qu'il récupère et modifie des vrais articles dans la base de données, et non plus nos articles statiques définis à la va-vite.

Pour cela, il y a très peu de modifications à réaliser : voici encore un exemple du code découpé que Symfony2 nous permet de réaliser ! En effet, il vous suffit de modifier les quatre endroits où on avait écrit un article en dur dans le contrôleur. Modifiez ces quatre endroits en utilisant bien le repository de l'entité Article, seules les méthodes `findAll()` et `find()` vont nous

servir pour le moment.

Attention, je vous demande également de faire attention au cas où l'article demandé n'existe pas. Si on essaie d'aller à la page /blog/article/4 alors que l'article d'id 4 n'existe pas, je veux une erreur correctement gérée ! On a déjà vu le déclenchement d'une erreur 404 lorsque le paramètre page de la page d'accueil n'était pas valide, reprenez ce comportement.

À la fin le contrôleur ne sera pas entièrement opérationnel, car il nous manque toujours la gestion des formulaires. Mais il sera déjà mieux avancé !

Et bien sûr, n'hésitez pas à nettoyer tous les codes de tests qu'on a pu utiliser lors de cette partie pour jouer avec les entités, maintenant on doit avoir un vrai contrôleur qui ne fait que son rôle.

Pratique

Il n'y a vraiment rien de compliqué dans notre nouveau contrôleur, le voici :

Code : PHP

```
<?php

// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sdz\BlogBundle\Entity\Article;

class BlogController extends Controller
{
    public function indexAction($page)
    {
        // On ne sait pas combien de pages il y a
        // Mais on sait qu'une page doit être supérieure ou égale à 1
        // Bien sûr pour le moment on ne se sert pas (encore !) de
        cette variable
        if ($page < 1) {
            // On déclenche une exception NotFoundHttpException
            // Cela va afficher la page d'erreur 404
            // On pourra la personnaliser plus tard
            throw $this->createNotFoundException('Page inexistante (page =
'. $page.')');
        }

        // Pour récupérer la liste de tous les articles : on utilise
        findAll()
        $articles = $this->getDoctrine()
            ->getManager()
            ->getRepository('SdzBlogBundle:Article')
            ->findAll();

        // L'appel de la vue ne change pas
        return $this->render('SdzBlogBundle:Blog:index.html.twig',
array(
    'articles' => $articles
));
    }

    public function voirAction($id)
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
            ->getManager();

        // Pour récupérer un article unique : on utilise find()
        $article = $em->getRepository('SdzBlogBundle:Article')
            ->find($id);
```

```
if ($article === null) {
    throw $this->createNotFoundException('Article[id='.$id.'] inexistant');
}

// On récupère les articleCompetence pour l'article $article
$liste_articleCompetence = $em->getRepository('SdzBlogBundle:ArticleCompetence')
    ->findByArticle($article->getId());

// Puis modifiez la ligne du render comme ceci, pour prendre en compte les variables :
return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
    'article' => $article,
    'liste_articleCompetence' => $liste_articleCompetence,
    // Pas besoin de passer les commentaires à la vue, on pourra y accéder via {{ article.commentaires }}
    // 'liste_commentaires' => $article->getCommentaires()
));
}

public function ajouterAction()
{
    // La gestion d'un formulaire est particulière, mais l'idée est la suivante :

    if ($this->get('request')->getMethod() == 'POST') {
        // Ici, on s'occupera de la création et de la gestion du formulaire

        $this->get('session')->setFlashBag()->add('info', 'Article bien enregistré');

        // Puis on redirige vers la page de visualisation de cet article
        return $this->redirect( $this->generateUrl('sdzblog_voir', array('id' => 1)) );
    }

    // Si on n'est pas en POST, alors on affiche le formulaire
    return $this->render('SdzBlogBundle:Blog:ajouter.html.twig');
}

public function modifierAction($id)
{
    // On récupère l'EntityManager
    $em = $this->getDoctrine()
        ->getEntityManager();

    // On récupère l'entité correspondant à l'id $id
    $article = $em->getRepository('SdzBlogBundle:Article')
        ->find($id);

    // Si l'article n'existe pas, on affiche une erreur 404
    if ($article == null) {
        throw $this->createNotFoundException('Article[id='.$id.'] inexistant');
    }

    // Ici, on s'occupera de la création et de la gestion du formulaire

    return $this->render('SdzBlogBundle:Blog:modifier.html.twig',
array(
    'article' => $article
));
}
```

```

public function supprimerAction($id)
{
    // On récupère l'EntityManager
    $em = $this->getDoctrine()
        ->getEntityManager();

    // On récupère l'entité correspondant à l'id $id
    $article = $em->getRepository('SdzBlogBundle:Article')
        ->find($id);

    // Si l'article n'existe pas, on affiche une erreur 404
    if ($article == null) {
        throw $this->createNotFoundException('Article[id='.$id.']'
inexistant');
    }

    if ($this->get('request')->getMethod() == 'POST') {
        // Si la requête est en POST, on supprimera l'article

        $this->get('session')->setFlashBag()->add('info', 'Article
bien supprimé');

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

    // Si la requête est en GET, on affiche une page de
confirmation avant de supprimer
    return $this->render('SdzBlogBundle:Blog:supprimer.html.twig',
array(
    'article' => $article
));
}

public function menuAction($nombre)
{
    $liste = $this->getDoctrine()
        ->getManager()
        ->getRepository('SdzBlogBundle:Article')
        ->findBy(
            array(),           // Pas de critère
            array('date' => 'desc'), // On trie par date
décroissante
            $nombre,             // On sélectionne $nombre
articles
            0                   // À partir du premier
);

    return $this->render('SdzBlogBundle:Blog:menu.html.twig', array(
        'liste_articles' => $liste // C'est ici tout l'intérêt : le
contrôleur passe les variables nécessaires au template !
    ));
}
}

```

Amélioration du contrôleur

Le contrôleur qu'on vient de faire n'est pas encore parfait, on peut faire encore mieux !

Je pense notamment à deux points en particulier :

- Le premier est l'utilisation implicite d'un ParamConverter pour éviter de vérifier à chaque fois l'existence d'un article (l'erreur 404 qu'on déclenche le cas échéant) ;
- Le deuxième est la pagination des articles sur la page d'accueil.

L'utilisation d'un ParamConverter

Le ParamConverter est une notion très simple : c'est un convertisseur de paramètre. Vous voilà bien avancés, n'est-ce pas ? 

Plus sérieusement, c'est vraiment cela. Il existe [un chapitre dédié](#) sur l'utilisation du ParamConverter dans la partie astuces. Mais sachez qu'il va convertir le paramètre en entrée du contrôleur dans une autre forme. Typiquement, nous avons le paramètre `$id` en entrée de certaines de nos actions, que nous voulons transformer directement en entité Article. Pour cela, rien de plus simple, il faut modifier la définition des méthodes du contrôleur comme suit :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// La définition :

public function voirAction($id)

// Devient :

public function voirAction(Article $article)
```



Bien entendu, n'oubliez pas de rajouter en début de fichier `use Sdz\BlogBundle\Entity\Article;` afin de pouvoir utiliser Article.

Et grâce aux mécanismes internes de Symfony2, vous retrouvez directement dans la variable `$article` l'article correspondant à l'id `$id`. Cela nous permet de supprimer l'utilisation du repository pour récupérer l'article, mais également le test de l'existence de l'article. Ces deux points sont fait automatiquement par le ParamConverter de Doctrine, et cela simplifie énormément nos méthodes. Voyez par vous-mêmes ce que devient la méthode `voirAction` du contrôleur :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function voirAction(Article $article)
{
    // À ce stade, la variable $article contient une instance de la
    // classe Article
    // Avec l'id correspondant à l'id contenu dans la route !

    // On récupère ensuite les articleCompetence pour l'article
    $article
        // On doit le faire à la main pour l'instant, car la relation
        // est unidirectionnelle
        // C'est-à-dire que $article->getArticleCompetences() n'existe
        // pas !
        $listeArticleCompetence = $this->getDoctrine()
            ->getManager()
            -
            >getRepository('SdzBlogBundle:ArticleCompetence')
                ->findByArticle($article-
                    >getId()));

    return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
        'article'          => $article,
        'listeArticleCompetence'  => $listeArticleCompetence
    ));
}
```

Merveilleux, non ? La méthode est réduite à son strict minimum, sans rien enlever en termes de fonctionnalité. Essayez avec un article qui n'existe pas : [/blog/article/5123123](#) ; nous avons bien une erreur 404.

Utiliser une jointure pour récupérer les articles

Actuellement sur la page d'accueil, avec l'action `indexAction()`, on ne récupère que les articles en eux-mêmes. Comme on en a parlé dans les précédents chapitres, cela veut dire que dans la boucle pour afficher les articles on ne peut pas utiliser les informations sur les relations (dans notre cas, les attributs `$image`, `$categories` et `$commentaires`). Enfin, on peut bien entendu les utiliser via `$article->getImage()`, mais dans ce cas une requête sera générée pour aller récupérée l'image... à chaque itération de la boucle sur les articles !

Ce comportement est bien sûr à proscrire, car le nombre de requêtes SQL va monter en flèche et ce n'est pas bon du tout pour les performances. Il faut donc modifier la requête initiale qui récupère les articles, pour y rajouter des jointures qui vont récupérer en une seule requête les articles ainsi que leurs entités jointes.

Tout d'abord, on va créer une méthode `getArticles()` dans le repository de l'entité `Article`, une version toute simple qui ne fait que récupérer les entités ordonnées :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ArticleRepository extends EntityRepository
{
    public function getArticles()
    {
        $query = $this->createQueryBuilder('a')
            ->orderBy('a.date', 'DESC')
            ->getQuery();

        return $query->getResult();
    }
}
```

Adaptions ensuite le contrôleur pour utiliser cette nouvelle méthode :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

class BlogController extends Controller
{
    public function indexAction($page)
    {
        // On ne sait pas combien de pages il y a
        // Mais on sait qu'une page doit être supérieure ou égale à 1
        // Bien sûr pour le moment on ne se sert pas (encore !) de
        // cette variable
        if ($page < 1) {
            // On déclenche une exception NotFoundHttpException
            // Cela va afficher la page d'erreur 404
            // On pourra la personnaliser plus tard
            throw $this->createNotFoundException('Page inexistante (page =
                '.$page.')');
        }
    }
}
```

```

    // Pour récupérer la liste de tous les articles : on utilise
    notre nouvelle méthode
    $articles = $this->getDoctrine()
        ->getManager()
        ->getRepository('SdzBlogBundle:Article')
            ->getArticles();

    // L'appel de la vue ne change pas
    return $this->render('SdzBlogBundle:Blog:index.html.twig',
array(
    'articles' => $articles
));
}

// ...
}

```

Maintenant, il nous faut mettre en place les jointures dans la méthode `getArticles()`, afin de charger toutes les informations sur les articles et éviter les dizaines de requêtes supplémentaires.

Dans notre exemple, nous allons afficher les données de l'entité `image` et des entités `categories` liées à chaque article. Il nous faut donc rajouter les jointures sur ces deux entités. On a déjà vu comment faire ces jointures, n'hésitez pas à essayer de les faire de votre côté avant de regarder ce code :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

// ...

public function getArticles($nombreParPage, $page)
{
    $query = $this->createQueryBuilder('a')
        // On joint sur l'attribut image
        ->leftJoin('a.image', 'i')
            ->addSelect('i')
        // On joint sur l'attribut categories
        ->leftJoin('a.categories', 'c')
            ->addSelect('c')
        ->orderBy('a.date', 'DESC')
        ->getQuery();

    return $query->getResult();
}

```

Comme vous pouvez le voir, les jointures se font simplement en utilisant les attributs existants de l'entité racine, ici l'entité `Article`. On rajoute donc juste les `leftJoin()` et les `addSelect()`, afin que Doctrine n'oublie pas de sélectionner les données qu'il joint. C'est tout ! Vous pouvez maintenant utiliser un `$article->getImage()` sans déclencher de nouvelle requête.

 Soit dit en passant, ces jointures peuvent justifier la mise en place d'une relation bidirectionnelle. En effet, dans l'état actuel on ne peut pas récupérer les informations des compétences liées à un article par exemple, car l'entité `Article` n'a pas d'attribut `ArticleCompetences`, donc pas de `->leftJoin()` possible. C'est l'entité `ArticleCompetence` qui est propriétaire de la relation unidirectionnelle. Si vous voulez afficher les compétences, vous devez commencer par rendre la relation bidirectionnelle. N'hésitez pas à le faire, c'est un bon entraînement !

La pagination des articles sur la page d'accueil

Paginer manuellement les résultats d'une requête n'est pas trop compliqué, il faut juste faire un peu de mathématiques à l'aide des variables suivantes :

- Nombre total d'articles ;
- Nombre d'articles à afficher par page ;
- La page courante.

Cependant, c'est un comportement assez classique et en bon développeur que nous sommes, trouvons une méthode plus simple et déjà prête ! Il existe en effet un paginateur intégré dans Doctrine2, qui permet de faire tout cela très simplement. Intégrons-le dans notre méthode `getArticles()` comme ceci :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\EntityRepository;
use Doctrine\ORM\Tools\Pagination\Paginator;

class ArticleRepository extends EntityRepository
{
    // On ajoute deux arguments : le nombre d'articles par page,
    // ainsi que la page courante
    public function getArticles($nombreParPage, $page)
    {
        // On déplace la vérification du numéro de page dans cette
        // méthode
        if ($page < 1) {
            throw new \InvalidArgumentException('L\'argument $page ne peut
être inférieur à 1 (valeur : "'. $page .'")');
        }

        // La construction de la requête reste inchangée
        $query = $this->createQueryBuilder('a')
            ->leftJoin('a.image', 'i')
            ->addSelect('i')
            ->leftJoin('a.categories', 'cat')
            ->addSelect('cat')
            ->orderBy('a.date', 'DESC')
            ->getQuery();

        // On définit l'article à partir duquel commencer la liste
        $query->setFirstResult((($page-1) * $nombreParPage))
        // Ainsi que le nombre d'articles à afficher
        ->setMaxResults($nombreParPage);

        // Enfin, on retourne l'objet Paginator correspondant à la
        // requête construite
        // (n'oubliez pas le use correspondant en début de fichier)
        return new Paginator($query);
    }
}
```

Maintenant que cette méthode est fonctionnelle pour la pagination, je vous invite à adapter le contrôleur pour prendre en compte cette pagination. Il faut donc utiliser correctement la méthode du repository que l'on vient de détailler, mais également donner à la vue toutes les données dont elle a besoin pour afficher une liste des pages existantes.



Au travail ! Encore une fois, faites l'effort de la réaliser de votre côté. Évidemment, je vous donne la solution, mais si vous n'avez pas essayé de chercher, vous ne progresserez pas. Courage !

Il existe bien entendu différentes manières de le faire, mais voici le code du contrôleur que je vous propose :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function indexAction($page)
{
    $articles = $this->getDoctrine()
        ->getManager()
        ->getRepository('SdzBlogBundle:Article')
        ->getArticles(3, $page); // 3 articles par page
: c'est totalement arbitraire !

// On ajoute ici les variables page et nb_page à la vue
return $this->render('SdzBlogBundle:Blog:index.html.twig',
array(
    'articles' => $articles,
    'page' => $page,
    'nombrePage' => ceil(count($articles) / 3)
));
}
```

C'est tout ! En effet, rappelez-vous l'architecture MVC, toute la logique de récupération des données est dans la couche Modèle : ici notre repository. Notre contrôleur est donc réduit à son strict minimum ; la couche Modèle, grâce à un Doctrine2 généreux en fonctionnalités, fait tout le travail.

Attention à une petite subtilité. Ici, la variable `$articles` contient une instance de `Paginator`. Concrètement, c'est une liste d'articles, dans notre cas une liste de 3 articles (on a mis cette valeur en dur). Vous pouvez l'utiliser avec un simple `foreach` par exemple. Cependant, pour obtenir le nombre de pages vous voyez qu'on a utilisé un `count($articles)` : ce `count` ne retourne pas 3, mais le nombre total d'articles dans la base de données ! Cela est possible avec les objets qui implémentent l'interface `Countable` de PHP.

Enfin, il nous reste juste la vue à adapter. Voici ce que je peux vous proposer, j'ai juste rajouté l'affichage de la liste des pages possibles :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #-}

{# ... le reste de la vue #-}

<div class="pagination">
    <ul>
        {# On utilise la fonction range(a, b) qui crée un tableau de
valeurs entre a et b #}
        {%- for p in range(1, nombrePage) %} 
            <li{%- if p == page %} class="active"{%- endif %}>
                <a href="{{ path('sdzblog_accueil', {'page': p}) }}>{{ p
}}</a>
            </li>
        {%- endfor %}
    </ul>
</div>
```

Ce qui donne le résultat visible à la figure suivante.



Le blog

[Accueil du blog](#)

[Ajouter un article](#)

Blog

Liste des articles

- [Test par dfg, le 01/01/2007](#)
- [Lala par winzou, le 01/01/2007](#)

1 2 3

The sky's the limit © 2012 and beyond.

Nos articles et la pagination s'affichent

Bingo !

Améliorer les vues

Dans cette section, on s'est surtout intéressés au contrôleur. Mais on peut également améliorer nos vues, en les mutualisant pour certaines actions par exemple. L'idée que j'ai en tête, c'est d'avoir une unique vue `article.html.twig` qu'on pourra utiliser sur la page d'accueil ainsi que sur la page d'un article. Bien sûr, ce n'est valable que si vous voulez la même présentation pour les deux, ce qui est mon cas.

Essayez donc de créer un `article.html.twig` générique, et de l'inclure depuis les deux autres vues.

`Article.html.twig`

Dans cette vue générique, on va intégrer l'affichage des informations de l'image et des catégories dont on a fait la jointure au début de cette section. Voici ma version :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/article.html.twig #-}

{# On utilise une variable temporaire, qu'on définit à false si
elle n'est pas déjà définie #}
{% set accueil = accueil|default(false) %}

<h2>
{# On vérifie qu'une image est bien associée à l'article #}
{% if article.image is not null %}

{% endif %}

{# Si on est sur la page d'accueil, on fait un lien vers
l'article, sinon non #}
{% if accueil %}
<a href="{{ path('sdzblog_voir', {'id': article.id}) }}>{{ article.titre }}</a>
```

```

    {%
      else %}
      {{ article.titre }}
    {%
      endif %}
  </h2>

<i>Le {{ article.date|date('d/m/Y') }}, par {{ article.auteur }}.</i>

<div class="well">
  {{ article.contenu }}
</div>

{# On affiche les catégories éventuelles #}
{%
  if article.categories.count > 0 %}
  <div class="well well-small">
    <p><i>
      Catégories :
      {%
        for categorie in article.categories %}
        {{ categorie.nom }}{%
          if not loop.last %}, {%
        endif %}
      {%
        endfor %}
    </i></p>
  </div>
{%
  endif %}

```

Ce que vous pouvez faire pour améliorer cette vue :

- Utiliser le slug plutôt que l'id pour les liens vers les articles, histoire d'avoir un meilleur référencement ;
- Utiliser un [bundle de Markdown](#) pour mettre en forme le texte du contenu de l'article ;
- Utiliser les microdata pour améliorer la compréhension de votre page par les moteurs de recherche, notamment avec le format [Article](#).

N'hésitez pas à traiter ces points vous-mêmes pour vous entraîner et pour améliorer le rendu de votre blog !

[Index.html.twig](#)

Dans cette vue, il faut juste modifier la liste afin d'utiliser la vue précédente pour afficher chaque article. Voici ma version :

Code : HTML& Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #-}

{# ... La liste devient : #}
<ul>
  {%
    for article in articles %}
    {# On inclut la vue à chaque itération dans la boucle #}
    {%
      include "SdzBlogBundle:Blog:article.html.twig" with
    {'accueil': true} %}
    <hr />
  {%
    else %}
    <p>Pas (encore !) d'articles</p>
  {%
    endfor %}
</ul>

{# ... #}

```

Il faut toujours vérifier que la vue incluse aura les variables qu'elle attend. Ici, la vue `article.html.twig` utilise la variable `{{ article }}`, il faut donc :

- Que cette variable existe dans la vue qui l'inclut ;
- Ou que la vue qui l'inclut précise cette variable dans le tableau :
`{% include "..." with {'article': ...} %}`.

Nous sommes dans le premier cas, la variable `{ { article } }` est définie par le `for` sur la liste des articles. Par contre, la variable `accueil` est transmise via le `with`.

Voir.html.twig

Dans cette vue, il faut juste remplacer l'affichage en dur de l'article par l'inclusion de notre nouvelle vue. Voici ma version :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

{%- extends "SdzBlogBundle::layout.html.twig" %}

{%- block title %}
    Lecture d'un article - {{ parent() }}
{%- endblock %}

{%- block sdzblog_body %}

    {# Ici, on inclut la vue #}
    {%- include "SdzBlogBundle:Blog:article.html.twig" %}

<p>
    <a href="{{ path('sdzblog_accueil') }}" class="btn">
        <i class="icon-chevron-left"></i>
        Retour à la liste
    </a>
    <a href="{{ path('sdzblog_modifier', {'id': article.id}) }}" class="btn">
        <i class="icon-edit"></i>
        Modifier l'article
    </a>
    <a href="{{ path('sdzblog_supprimer', {'id': article.id}) }}" class="btn">
        <i class="icon-trash"></i>
        Supprimer l'article
    </a>
</p>

{%- endblock %}
```

Pour conclure

Et voilà, le premier TP du tutoriel s'achève ici. J'espère que vous avez pu exploiter toutes les connaissances que vous avez pu acquérir jusqu'ici, et qu'il vous a aidé à vous sentir plus à l'aise.

La prochaine partie du tutoriel va vous emmener plus loin avec Symfony2, pour connaître tous les détails qui vous permettront de créer votre site internet de toutes pièces. À bientôt !

En résumé

- Vous savez maintenant construire vos entités ;
- Vous savez maintenant développer un contrôleur abouti ;
- Vous savez maintenant faire des jointures et plus encore dans vos repositories ;
- Vous savez maintenant inclure habilement des vues pour éviter de dupliquer du code ;
- Bravo !

Partie 4 : Allons plus loin avec Symfony2

Créer des formulaires avec Symfony2

Quoi de plus important sur un site web que les formulaires ? En effet, les formulaires sont l'interface entre vos visiteurs et votre contenu. Chaque commentaire, chaque article de blog, etc., tous passent par l'intermédiaire d'un visiteur et d'un formulaire pour exister dans votre base de données.

L'objectif de ce chapitre est donc de vous donner enfin les outils pour créer efficacement ces formulaires grâce à la puissance du composant `Form` de Symfony2. Ce chapitre va de paire avec le prochain, dans lequel nous parlerons de la validation des données, celles que vos visiteurs vont entrer dans vos nouveaux formulaires.

Gestion des formulaires

L'enjeu des formulaires

Vous avez déjà créé des formulaires en HTML et PHP, vous savez donc que c'est une vraie galère ! À moins d'avoir créé vous-mêmes un système dédié, gérer correctement des formulaires s'avère être un peu mission impossible. Par « correctement », j'entends de façon maintenable, mais surtout **réutilisable**. Heureusement, le composant `Form` de Symfony2 arrive à la rescousse !



N'oubliez pas que les composants peuvent être utilisés hors d'un projet Symfony2. Vous pouvez donc reprendre le composant `Form` dans votre site même si vous n'utilisez pas Symfony2.

Un formulaire Symfony2, qu'est-ce que c'est ?

La vision Symfony2 sur les formulaires est la suivante : un formulaire se construit sur un objet existant, et son objectif est d'hydrater cet objet.

Un objet existant

Il nous faut donc des objets de base avant de créer des formulaires. Mais en fait, ça tombe bien : on les a déjà, ces objets ! En effet, un formulaire pour ajouter un article de blog va se baser sur l'objet `Article`, objet que nous avons construit lors du chapitre précédent. Tout est cohérent.



Je dis bien « objet » et non « entité Doctrine2 ». En effet, les formulaires n'ont pas du tout besoin d'une entité pour se construire, mais uniquement d'un simple objet. Heureusement, nos entités sont de simples objets avant d'être des entités, donc elles conviennent parfaitement. Je précise ce point pour vous montrer que les composants Symfony2 sont indépendants les uns des autres.

Pour la suite de ce chapitre, nous allons utiliser notre objet `Article`. C'est un exemple simple qui va nous permettre de construire notre premier formulaire. Je rappelle son code, sans les annotations pour plus de clarté (et parce qu'elles ne nous regardent pas ici) :

Code : PHP

```
<?php

namespace Sdz\BlogBundle\Entity;

use Sdz\BlogBundle\Entity\Tag;

class Article
{
    private $id;
    private $date;
    private $titre;
    private $contenu;
```

```

private $auteur;
private $publication;
private $image;
private $categories;
private $commentaires;

public function __construct()
{
    $this->date      = new \Datetime;
    $this->publication = true;
    $this->categories = new
\Doctrine\Common\Collections\ArrayCollection();
    $this->commentaires = new
\Doctrine\Common\Collections\ArrayCollection();
}

// ... Les getters et setters
}

```



Rappel : la convention pour le nom des *getters/setters* est importante : lorsque l'on parlera du champ « auteur », le composant Form utilisera l'objet via les méthodes `setAuteur()` et `getAuteur()` (comme le faisait Doctrine2 de son côté). Donc si vous aviez eu `set_auteur()` ou `recuperer_auteur()`, cela n'aurait pas fonctionné.

Objectif : hydrater cet objet

Hydrater ? Un terme précis pour dire que le formulaire va remplir les attributs de l'objet avec les valeurs entrées par le visiteur. Faire `<?php $article->setAuteur('winzou') ?>`, `<?php $article->setDate(new \Datetime())`, etc., c'est hydrater l'objet Article.

Le formulaire en lui-même n'a donc comme seul objectif que d'hydrater un objet. Ce n'est qu'une fois l'objet hydraté que vous pourrez en faire ce que vous voudrez : enregistrer en base de données dans le cas de notre objet Article, envoyer un e-mail dans le cas d'un objet Contact, etc. Le système de formulaire ne s'occupe pas de ce que vous faites de votre objet, il ne fait que l'hydrater.

Une fois que vous avez compris cela, vous avez compris l'essentiel. Le reste n'est que de la syntaxe à connaître.

Gestion basique d'un formulaire

Concrètement, pour créer un formulaire, il nous faut deux choses :

- Un objet (on a toujours notre objet Article) ;
- Un moyen pour construire un formulaire à partir de cet objet, un FormBuilder, « constructeur de formulaire » en français.

Pour faire des tests, placez-vous dans l'action `ajouterAction()` de notre contrôleur Blog et modifiez-la comme suit :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

use Sdz\BlogBundle\Entity\Article;

// ...

public function ajouterAction()
{
    // On crée un objet Article
    $article = new Article();

```

```

// On crée le FormBuilder grâce à la méthode du contrôleur
$formBuilder = $this->createFormBuilder($article);

// On ajoute les champs de l'entité que l'on veut à notre formulaire
$formBuilder
    ->add('date', 'date')
    ->add('titre', 'text')
    ->add('contenu', 'textarea')
    ->add('auteur', 'text')
    ->add('publication', 'checkbox');

// Pour l'instant, pas de commentaires, catégories, etc., on les gérera plus tard

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

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

```

Pour le moment, ce formulaire n'est pas opérationnel. On va pouvoir l'afficher, mais il ne se passera rien lorsqu'on le validera.

Mais avant cette étape, essayons de comprendre le code présenté. Dans un premier temps, on récupère le FormBuilder. Cet objet n'est pas le formulaire en lui-même, c'est un constructeur de formulaire. On lui dit : « Crée un formulaire autour de l'objet \$article », puis : « Ajoute les champs date, titre, contenu et auteur. » Et enfin : « Maintenant, donne-moi le formulaire construit avec tout ce que je t'ai dit auparavant. »

Prenons le temps de bien faire la différence entre les attributs de l'objet hydraté et les champs du formulaire. Un formulaire n'est pas du tout obligé d'hydrater tous les attributs d'un objet. On pourrait très bien ne pas utiliser le pseudo pour l'instant par exemple, et ne pas mettre de champ auteur dans notre formulaire. L'objet, lui, contient toujours l'attribut auteur, mais il ne sera juste pas hydraté par le formulaire. Bon, en l'occurrence, ce n'est pas le comportement que l'on veut (on va considérer le pseudo comme obligatoire pour un article), mais sachez que c'est possible. 😊 D'ailleurs, si vous avez l'œil, vous avez remarqué qu'on n'ajoute pas de champ id : comme il sera rempli automatiquement par Doctrine (grâce à l'auto-incrémentation), le formulaire n'a pas besoin de remplir cet attribut.

Enfin, c'est avec cet objet \$form généré que l'on pourra gérer notre formulaire : vérifier qu'il est valide, l'afficher, etc. Par exemple, ici, on utilise sa méthode <?php \$form->createView() qui permet à la vue d'afficher ce formulaire. Concernant l'affichage du formulaire, j'ai une bonne nouvelle pour vous : Symfony2 nous permet d'afficher un formulaire simple en une seule ligne HTML ! Si, si : rendez-vous dans la vue Blog/formulaire.html.twig et ajoutez ces quelques lignes là où nous avions laissé un trou :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/formulaire.html.twig #-}

<h3>Formulaire d'article</h3>

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

```

Ensuite, admirez le résultat à l'adresse suivante : http://localhost/Symfony/web/app_dev.php/blog/ajouter. Impressionnant, non ?

Ajouter un article

Formulaire d'article

Date

2012 12 -
20

Titre

Contenu

Auteur

Publication

Valider

Aperçu du formulaire simple

Vous ajoutez un nouvel article.

Grâce à la fonction Twig {{ form_widget() }}, on peut afficher un formulaire entier en une seule ligne. Alors bien sûr, il n'est pas forcément à votre goût pour le moment, mais voyez le bon côté des choses : pour l'instant, on est en plein développement, on veut tester notre formulaire. On s'occupera de l'esthétique plus tard. N'oubliez pas également de rajouter les balises <form> HTML et le bouton de soumission, car la fonction n'affiche que l'intérieur du formulaire.

Bon, évidemment, comme je vous l'ai dit, ce code ne fait qu'afficher le formulaire, il n'est pas encore question de gérer sa soumission. Mais patience, on y arrive.



La date sélectionnée par défaut est celle d'aujourd'hui, et la checkbox « Publication » est déjà cochée : comment est-ce possible ?

Bonne question ! Il est important de savoir que ces deux points ne sont pas là par magie, et que dans Symfony2 tout est cohérent. Si vous vous rappelez, on avait défini des valeurs dans le constructeur de l'entité Article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

// ...

public function __construct()
{
    $this->date      = new \Datetime;
    $this->publication = true;
// ...
```

```
}
```

C'est à ce moment qu'est définie la valeur de ces deux attributs, et c'est sur la valeur de ces attributs que se base le formulaire pour remplir ses champs. Voilà l'origine de ces valeurs !

Ajouter des champs

Vous pouvez le voir, ajouter des champs à un formulaire se fait assez facilement avec la méthode `<?php $formBuilder->add()` du `FormBuilder`. Les arguments sont les suivants :

1. Le nom du champ ;
2. Le type du champ ;
3. Les options du champ, sous forme de tableau.

Par « type de champ », il ne faut pas comprendre « type HTML » comme `text`, `password` ou `select`. Il faut comprendre « type sémantique ». Par exemple, le type `date` que l'on a utilisé affiche trois champs `select` à la suite pour choisir le jour, le mois et l'année. Il existe aussi un type `timezone` pour choisir le fuseau horaire. Bref, il en existe pas mal et ils n'ont rien à voir avec les types HTML, ils vont bien plus loin que ces derniers ! N'oubliez pas, Symfony2 est magique ! 

Voici l'ensemble des types de champ disponibles. Je vous dresse ici la liste avec pour chacun un lien vers la documentation : allez-y à chaque fois que vous avez besoin d'utiliser tel ou tel type.

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



Ayez bien cette liste en tête : le choix d'un type de champ adapté à l'attribut de l'objet sous-jacent est une étape importante dans la création d'un formulaire.

Il est primordial de bien faire correspondre les types de champ du formulaire avec les types d'attributs que contient votre objet. En effet, si le formulaire retourne un booléen alors que votre objet attend du texte, ils ne vont pas s'entendre. La figure suivante montre donc comment on a choisi les types de champ de formulaire selon l'objet `Article`.

Constructeur de formulaire FormBuilder :

```
$formBuilder
    ->add('date', 'date')
    ->add('titre', 'text')
    ->add('contenu', 'textarea')
    ->add('auteur', 'text')
    ->add('publication', 'checkbox');
```

Classe Article liée au formulaire :

```
class Article
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

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

    /** @ORM\Column(name="titre", type="string", length=255) */
    private $titre;

    /** @ORM\Column(name="auteur", type="string", length=255) */
    private $auteur;

    /** @ORM\Column(name="publication", type="boolean") */
    private $publication;

    /** @ORM\Column(name="contenu", type="text") */
    private $contenu;

    /** @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image") */
    private $image;

    /** @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Categorie") */
    private $categories;

    /** @ORM\OneToMany(targetEntity="Sdz\BlogBundle\Entity\Commentaire") */
    private $commentaires;
```

Correspondances entre le formulaire et l'objet

Gestion de la soumission d'un formulaire

Afficher un formulaire c'est bien, mais faire quelque chose lorsqu'un visiteur le soumet, c'est quand même mieux !

- Pour gérer l'envoi du formulaire, il faut tout d'abord vérifier que la requête est de type POST : cela signifie que le visiteur est arrivé sur la page en cliquant sur le bouton submit du formulaire. Lorsque c'est le cas, on peut traiter notre formulaire.
- Ensuite, il faut faire le lien entre les variables de type POST et notre formulaire, pour que les variables de type POST viennent remplir les champs correspondants du formulaire. Cela se fait via la méthode bind() du formulaire. Cette

méthode dit au formulaire : « Voici la requête d'entrée (nos variables de type POST entre autres). Lis cette requête, récupère les valeurs qui t'intéressent et hydrate l'objet. » Comme vous pouvez le voir, elle fait beaucoup de choses !

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

Ce n'est qu'après ces trois étapes que l'on peut traiter notre objet hydraté : sauvegarder en base de données, envoyer un e-mail, etc.

Vous êtes un peu perdus ? C'est parce que vous manquez de code. Voici comment faire tout ce que l'on vient de dire, dans le contrôleur :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

use Sdz\BlogBundle\Entity\Article;

// ...

public function ajouterAction()
{
    $article = new Article;

    // J'ai raccourci cette partie, car c'est plus rapide à écrire !
    $form = $this->createFormBuilder($article)
        ->add('date', 'date')
        ->add('titre', 'text')
        ->add('contenu', 'textarea')
        ->add('auteur', 'text')
        ->add('publication', 'checkbox')
        ->getForm();

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

    // On vérifie qu'elle est de type POST
    if ($request->getMethod() == 'POST') {
        // On fait le lien Requête <-> Formulaire
        // À partir de maintenant, la variable $article contient les
        // valeurs entrées dans le formulaire par le visiteur
        $form->bind($request);

        // On vérifie que les valeurs entrées sont correctes
        // (Nous verrons la validation des objets en détail dans le
        // prochain chapitre)
        if ($form->isValid()) {
            // On l'enregistre notre objet $article dans la base de
            // données
            $em = $this->getDoctrine()->getManager();
            $em->persist($article);
            $em->flush();

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

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

    return $this->render('SdzBlogBundle:Blog:ajouter.html.twig',
```

```

array(
    'form' => $form->createView(),
);
}

```

Si le code paraît long, c'est parce que j'ai mis plein de commentaires ! Prenez le temps de bien le lire et de bien le comprendre : vous verrez, c'est vraiment simple. N'hésitez pas à le tester. Essayez de ne pas remplir un champ pour observer la réaction de Symfony2. Vous voyez que ce formulaire gère déjà très bien les erreurs, il n'enregistre l'article que lorsque tout va bien.



N'hésitez pas à tester votre formulaire en ajoutant des articles ! Il est opérationnel, et les articles que vous ajoutez sont réellement enregistrés en base de données.

Si vous l'avez bien testé, vous vous êtes rendu compte qu'on est obligés de cocher le champ `publication`. Ce n'est pas tellement le comportement voulu, car on veut pouvoir enregistrer un article sans forcément le publier (pour finir la rédaction plus tard par exemple). Pour cela, nous allons utiliser le troisième argument de la méthode `$formBuilder->add()` qui correspond aux options du champ. Les options se présentent sous la forme d'un simple tableau. Pour rendre le champ facultatif, il faut définir l'option `required` à `false`, comme suit :

Code : PHP

```

<?php
$formBuilder->add('publication', 'checkbox', array('required' =>
false));

```

Rappelez-vous donc : un champ de formulaire est requis par défaut. Si vous voulez le rendre facultatif, vous devez préciser l'option `required` à la main.

Un mot également sur la validation que vous rencontrez depuis le navigateur : impossible de valider le formulaire si un champ obligatoire n'est pas rempli :

The screenshot shows a web form with two fields: "Auteur" and "Publication". The "Auteur" field contains the value "winzou". The "Publication" field has a red border around its input box, indicating it is invalid. Below the input box, a yellow callout box contains the text: "Veuillez cocher cette case si vous souhaitez continuer." To the right of the form, the text "Le champ « Publication » était obligatoire" is displayed.

Vous ajoutez un nouvel article.

Pourtant, nous n'avons pas utilisé de JavaScript ! C'est juste du HTML5. En mettant l'attribut `required="required"` à une balise `<input>`, le navigateur interdit la validation du formulaire tant que cet input est vide. Pratique ! Mais attention, cela n'empêche pas de faire une validation côté serveur, au contraire. En effet, si quelqu'un utilise votre formulaire avec un vieux navigateur qui ne supporte pas le HTML5, il pourra valider le formulaire sans problème.

Gérer les valeurs par défaut du formulaire

L'un des besoins courants dans les formulaires, c'est de mettre des valeurs prédéfinies dans les champs. Cela peut servir pour des valeurs par défaut (préremplir la date, par exemple) ou alors lors de l'édition d'un objet déjà existant (pour l'édition d'un article, on souhaite remplir le formulaire avec les valeurs de la base de données).

Heureusement, cela se fait très facilement. Il suffit de modifier l'instance de l'objet, ici `$article`, avant de le passer en argument à la méthode `createFormBuilder`, comme ceci :

Code : PHP

```
<?php
// On crée un nouvel article
$article = new Article;

// Ici, on préremplit avec la date d'aujourd'hui, par exemple
// Cette date sera donc préaffichée dans le formulaire, cela
// facilite le travail de l'utilisateur
$article->setDate(new \Datetime());

// Et on construit le formBuilder avec cette instance d'article
$formBuilder = $this->createFormBuilder($article);

// N'oubliez pas d'ajouter les champs comme précédemment avec la
// méthode ->add()
```

Et si vous voulez modifier un article déjà enregistré en base de données, alors il suffit de le récupérer avant la création du formulaire, comme ceci :

Code : PHP

```
<?php
// Récupération d'un article déjà existant, d'id $id.
$article = $this->getDoctrine()
    ->getRepository('Sdz\BlogBundle\Entity\Article')
    ->find($id);

// Et on construit le formBuilder avec cette instance d'article,
// comme précédemment
$formBuilder = $this->createFormBuilder($article);

// N'oubliez pas d'ajouter les champs comme précédemment avec la
// méthode ->add()
```

Personnaliser l'affichage d'un formulaire

Jusqu'ici, nous n'avons pas du tout personnalisé l'affichage de notre formulaire. Voyez quand même le bon côté des choses : on travaillait côté PHP, on a pu avancer très rapidement sans se soucier d'écrire les balises `<input>` à la main, ce qui est long et sans intérêt.

Mais bon, à un moment donné, il faut bien mettre la main à la pâte et faire des formulaires dans le même style que son site. Pour cela, je ne vais pas m'étendre, mais voici un exemple qui vous permettra de faire à peu près tout ce que vous voudrez :

Code : HTML & Django

```
<form action="{{ path('votre_route') }}" method="post" {{ form_enctype(form) }}>

    # Les erreurs générales du formulaire. #
    {{ form_errors(form) }}

    <div>
        # Génération du label. #
        {{ form_label(form.titre, "Titre de l'article") }}

        # Affichage des erreurs pour ce champ précis. #

```

```
{ { form_errors(form.titre) } }

{# Génération de l'input. #}
{ { form_widget(form.titre) } }

</div>

{# Idem pour un autre champ. #}
<div>
{ { form_label(form.contenu, "Contenu de l'article") } }
{ { form_errors(form.contenu) } }
{ { form_widget(form.contenu) } }
</div>

{# Génération des champs pas encore écrits.
Dans cet exemple, ce serait « date », « auteur » et « publication »
,
mais aussi le champ CSRF (géré automatiquement par Symfony !)
et tous les champs cachés (type « hidden »). #}
{ { form_rest(form) } }

</form>
```

Pour plus d'information concernant l'habillage des formulaires, je vous invite à consulter la [documentation à ce sujet](#). Cela s'appelle en anglais le *form theming*.



Qu'est-ce que le CSRF ?

Le champ CSRF, pour *Cross Site Request Forgeries*, permet de vérifier que l'internaute qui valide le formulaire est bien celui qui l'a affiché. C'est un moyen de se protéger des envois de formulaire frauduleux ([plus d'informations sur le CSRF](#)). C'est un champ que Symfony2 rajoute automatiquement à tous vos formulaires, afin de les sécuriser sans même que vous vous en rendiez compte. 😊

Créer des types de champ personnalisés

Il se peut que vous ayez envie d'utiliser un type de champ précis, mais que ce type de champ n'existe pas par défaut. Heureusement, vous n'êtes pas coincés, vous pouvez vous en sortir en créant votre propre type de champ. Vous pourrez ensuite utiliser ce champ comme n'importe quel autre dans vos formulaires.

Imaginons par exemple que vous n'aimiez pas le rendu du champ `date` avec ces trois balises `<select>` pour sélectionner le jour, le mois et l'année. Vous préféreriez un joli `datepicker` en JavaScript. La solution ? Créez un nouveau type de champ !

Je ne vais pas décrire la démarche ici, mais sachez que cela existe et que [la documentation traite ce point](#).

Externaliser la définition de ses formulaires

Vous savez enfin créer un formulaire. Ce n'était pas très compliqué, nous l'avons rapidement fait et ce dernier se trouve être assez joli. Mais vous souvenez-vous de ce que j'avais promis au début : nous voulions un formulaire *réutilisable* ; or là, tout est dans le contrôleur, et je vois mal comment le réutiliser ! Pour cela, il faut détacher la définition du formulaire dans une classe à part, nommée `ArticleType` (par convention).

Définition du formulaire dans ArticleType

`ArticleType` n'est pas notre formulaire. Comme tout à l'heure, c'est notre *constructeur de formulaire*. Par convention, on va mettre tous nos `xxxType.php` dans le répertoire `Form` du bundle. En fait, on va encore utiliser le générateur ici, qui sait générer les `FormType` pour nous, et vous verrez qu'on y gagne !

Exécutez donc la commande suivante :

Code : Console

```
php app/console doctrine:generate:form SdzBlogBundle:Article
```

Comme vous pouvez le voir c'est une commande Doctrine, car c'est lui qui a toutes les informations sur notre objet Article. Maintenant, vous pouvez aller voir le résultat dans le fichier `src/Sdz/BlogBundle/Form/ArticleType.php`.

On va commencer tout de suite par améliorer ce formulaire. En effet, vous pouvez voir que les types de champ ne sont pas précisés : le composant Form va les deviner à partir des annotations Doctrine qu'on a mis dans l'objet. Ce n'est pas une bonne pratique, car cela peut être source d'erreur, c'est pourquoi je vous invite dès maintenant à remettre explicitement les types comme on avait déjà fait dans le contrôleur :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Form/ArticleType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
->add('date', 'date')
->add('titre', 'text')
->add('contenu', 'textarea')
->add('auteur', 'text')
->add('publication', 'checkbox', array('required' => false))
    ;
}

public function setDefaultOptions(OptionsResolverInterface
$resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Sdz\BlogBundle\Entity\Article'
    ));
}

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



J'ai également supprimé les champs `image` et `categories`, que nous verrons différemment plus loin dans ce chapitre.

Comme vous pouvez le voir, on n'a fait que déplacer la construction du formulaire, du contrôleur à une classe externe. Cet ArticleType correspond donc en fait à la définition des champs de notre formulaire. Ainsi, si l'on utilise le même formulaire sur plusieurs pages différentes, on utilisera ce même ArticleType. Fini le copier-coller ! Voici la réutilisabilité. 😊

Rappelez-vous également, un formulaire se construit autour d'un objet. Ici, on a indiqué à Symfony2 quel était cet objet grâce à la méthode `setDefaultOptions()`, dans laquelle on a défini l'option `data_class`.

Le contrôleur épuré

Avec cet ArticleType, la construction du formulaire côté contrôleur s'effectue grâce à la méthode `createForm()` du contrôleur (et non plus `createFormBuilder()`). Cette méthode utilise le composant `Form` pour construire un formulaire à partir du ArticleType passé en argument. Depuis le contrôleur, on récupère donc directement un formulaire, on ne passe plus par le constructeur de formulaire comme précédemment. Voyez par vous-mêmes :

Code : PHP

```
<?php
// Dans le contrôleur

$article = new Article;
$form = $this->createForm(new ArticleType, $article);
```

En effet, si l'on s'est donné la peine de créer un objet à l'extérieur du contrôleur, c'est pour que ce contrôleur soit plus simple. C'est réussi !

Au final, en utilisant cette externalisation et en supprimant les commentaires, voici à quoi ressemble la gestion d'un formulaire dans Symfony2 :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

use Sdz\BlogBundle\Entity\Article;
// N'oubliez pas d'ajouter le ArticleType
use Sdz\BlogBundle\Form\ArticleType;

// ...

public function ajouterAction()
{
    $article = new Article;
    $form = $this->createForm(new ArticleType, $article);

    $request = $this->get('request');
    if ($request->getMethod() == 'POST') {
        $form->bind($request);

        if ($form->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->persist($article);
            $em->flush();

            return $this->redirect($this->generateUrl('sdzblog_accueil'));
        }
    }

    return $this->render('SdzBlogBundle:Blog:ajouter.html.twig',
array(
    'form' => $form->createView(),
));
}
```

Plutôt simple, non ? Au final, votre code métier, votre code qui fait réellement quelque chose, se trouve là où l'on a utilisé l'EntityManager. Pour l'exemple, nous n'avons fait qu'enregistrer l'article en base de données, mais c'est ici que vous pourrez envoyer un e-mail, ou effectuer toute autre action dont votre site internet aura besoin.

Les formulaires imbriqués Intérêt de l'imbrication



Pourquoi imbriquer des formulaires ?

C'est souvent le cas lorsque vous avez des relations entre vos objets : vous souhaitez ajouter un objet A, mais en même temps un autre objet B qui est lié au premier. Exemple concret : vous voulez ajouter un client à votre application, votre Client est lié à une Adresse, mais vous avez envie d'ajouter l'adresse sur la même page que votre client, depuis le même formulaire. S'il fallait deux pages pour ajouter client puis adresse, votre site ne serait pas très ergonomique. Voici donc toute l'utilité de l'imbrication des formulaires !

Un formulaire est un champ

Eh oui, voici tout ce que vous devez savoir pour imbriquer des formulaires entre eux. Considérez un de vos formulaires comme un champ, et appelez ce simple champ depuis un autre formulaire ! Bon, facile à dire, mais il faut savoir le faire derrière.

D'abord, créez le formulaire de notre entité Image. Vous l'aurez compris, on peut utiliser le générateur ici, exécutez donc cette commande :

Code : Console

```
php app/console doctrine:generate:form SdzBlogBundle:Image
```

En explicitant les types des champs, cela donne le code suivant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Form/ImageType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ImageType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('url', 'text')
            ->add('alt', 'text')
        ;
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Sdz\BlogBundle\Entity\Image'
        ));
    }

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

Ensuite, il existe deux façons d'imbriquer ce formulaire :

1. Avec une relation simple où l'on imbrique une seule fois un sous-formulaire dans le formulaire principal. C'est le cas le plus courant, celui de notre Article avec une seule Image.
2. Avec une relation multiple, où l'on imbrique plusieurs fois le sous-formulaire dans le formulaire principal. C'est le cas d'un Client qui pourrait enregistrer plusieurs Adresse.

Relation simple : imbriquer un seul formulaire

C'est le cas le plus courant, qui correspond à notre exemple de l'Article et de son Image. Pour imbriquer un seul formulaire en étant cohérent avec une entité, il faut que l'entité du formulaire principal (ici, Article) ait une relation *One-To-One* ou *Many-To-One* avec l'entité (ici, Image) dont on veut imbriquer le formulaire.

Une fois que vous savez cela, on peut imbriquer nos formulaires. C'est vraiment simple : allez dans ArticleType et ajoutez un champ image (du nom de la propriété de notre entité), de type... ImageType, bien sûr !

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Form/ArticleType.php

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
            ->add('date', 'date')
            ->add('titre', 'text')
            ->add('contenu', 'textarea')
            ->add('auteur', 'text')
            ->add('publication', 'checkbox', array('required' => false))
            ->add('image', new ImageType()) // Rajoutez cette ligne
    ;
}
// ...
}
```

C'est tout ! Allez sur la page d'ajout : [/blog/ajouter](#). Le formulaire est déjà à jour (voir figure suivante), avec une partie « Image » où l'on peut remplir les deux seuls champs de ce formulaire, les champs « Url » et « Alt ». C'était d'une facilité déconcertante, n'est-ce pas ?

Ajouter un article

Formulaire d'article

Date

2012 10 -
04

Titre

Contenu

Auteur

Publication

Image

Url

Alt

Valider

Le formulaire est à jour

The screenshot shows a web-based form for creating an article. It includes fields for date (set to 2012-10-04), title, content, author, publication status (checked), and an image section (with URL and alt attributes). The 'Image' section is highlighted with a red border. A validation button is at the bottom, and a status message 'Le formulaire est à jour' is on the right.

Réfléchissons bien à ce qu'on vient de faire.

D'un côté, nous avons l'entité Article qui possède un attribut image. Cet attribut image contient, lui, un objet Image. Il ne peut pas contenir autre chose, à cause du setter associé : celui-ci force l'argument à être un objet de classe Image.

L'objectif du formulaire est donc de venir injecter dans cet attribut image un objet Image, et pas autre chose ! On l'a vu au début de ce chapitre, un formulaire de type XxxType retourne un objet de classe Xxx. Il est donc tout à fait logique de mettre dans ArticleType, un champ image de type ImageType.

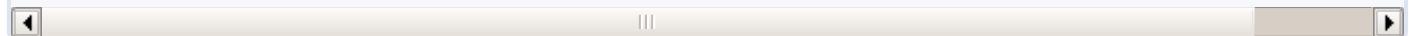
Sachez qu'il est bien entendu possible d'imbriquer les formulaires à l'infini de cette façon. La seule limitation, c'est de faire quelque chose de compréhensible pour vos visiteurs, ce qui est tout de même le plus important.

Si lorsque vous validez votre formulaire vous avez une erreur de ce type :

Code : Console

```
A new entity was found through the relationship 'Sdz\BlogBundle\Entity\Article#cate  
that was not configured to cascade persist operations for entity:  
Sdz\BlogBundle\Entity\Categorie@000000000579b29e0000000061a76c55. To solve this iss  
Either explicitly call EntityManager#persist() on this unknown entity or configure  
persist this association in the mapping for example @ManyToOne(...,cascade={"persist
```

```
you cannot find out which entity causes the problem implement  
'Sdz\BlogBundle\Entity\Categorie#__toString()' to get a clue.
```



... c'est que Doctrine ne sait pas quoi faire avec l'entité `Image` qui est dans l'entité `Article`, car vous ne lui avez pas dit de persister cette entité. Pour corriger l'erreur, il faut dire à Doctrine de persister cet objet `Image` bien sûr, suivez simplement les indications du message d'erreur :

- Soit vous ajoutez manuellement un `$em->persist($article->getImage())` dans le contrôleur, avant le `flush()` ;
- Soit vous ajoutez une option à l'annotation `@ORM\OneToOne` dans l'entité `Article`, ce que nous avons fait si vous suivez ce cours depuis le début, comme ceci :

Code : Autre

```
/**  
 * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image", cascade={"persist"})  
 */  
private $image;
```

C'est fini pour l'imbrication simple d'un formulaire dans un autre. Passons maintenant à l'imbrication multiple.

Relation multiple : imbriquer un même formulaire plusieurs fois

On imbrique un même formulaire plusieurs fois lorsque deux entités sont en relation *Many-To-One* ou *Many-To-Many*.

On va prendre l'exemple ici de l'imbrication de plusieurs `CategorieType` dans le `ArticleType` principal. Attention, cela veut dire qu'à chaque ajout d'`Article`, on aura la possibilité de créer de nouvelles `Categorie`. Ce n'est pas le comportement classique qui consiste plutôt à sélectionner des `Categorie` existantes. Ce n'est pas grave, c'est pour l'exemple, sachant que plus loin dans ce chapitre on étudie également la manière de sélectionner ces catégories.

Tout d'abord, créez le formulaire `CategorieType` grâce au générateur :

Code : Console

```
php app/console doctrine:generate:form SdzBlogBundle:Categorie
```

Voici ce que cela donne après avoir explicité les champs encore une fois :

Code : PHP

```
<?php  
// src/Sdz/Bundle/Form/CategorieType.php  
  
namespace Sdz\BlogBundle\Form;  
  
use Symfony\Component\Form\AbstractType;  
use Symfony\Component\Form\FormBuilderInterface;  
use Symfony\Component\OptionsResolver\OptionsResolver;  
  
class CategorieType extends AbstractType  
{  
    public function buildForm(FormBuilderInterface $builder, array  
    $options)
```

```

    {
        $builder
            ->add('nom', 'text') // Ici, explicitez le type du champ
        ;
    }

    public function setDefaultOptions(OptionsResolverInterface
$resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Sdz\BlogBundle\Entity\Categorie'
));
}

public function getName()
{
    return 'sdz_blogbundle_categorietype';
}
}

```

Maintenant, il faut rajouter le champ categories dans le ArticleType. Il faut pour cela utiliser le type collection et lui passer quelques options, comme ceci :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Form/ArticleType.php

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
            ->add('date', 'date')
            ->add('titre', 'text')
            ->add('contenu', 'textarea')
            ->add('auteur', 'text')
            ->add('publication', 'checkbox', array('required' => false))
            ->add('image', new ImageType())
            /*
        * Rappel :
        ** - 1er argument : nom du champ, ici « categories », car c'est le
        nom de l'attribut
        ** - 2e argument : type du champ, ici « collection » qui est une
        liste de quelque chose
        ** - 3e argument : tableau d'options du champ
        */
            ->add('categories', 'collection', array('type' => new
CategorieType(),
                'allow_add' =>
true,
                'allow_delete' =>
true))
            ;
    }

    // ...
}

```

On a ici utilisé le type de champ collection, qui permet en réalité de construire une collection (une liste) de n'importe quoi. Ici on a dit grâce à l'option type qu'il doit créer une liste de CategorieType, mais on aurait pu faire une liste de type text : le formulaire aurait donc injecté dans l'attribut categories un simple tableau de textes.

Ce champ de type `collection` comporte plusieurs options en plus du type. Vous notez les options `allow_add` et `allow_delete`, qui autorisent au formulaire d'ajouter des entrées en plus dans la collection, ainsi que d'en supprimer. En effet, on pourrait tout à fait ne pas autoriser ces actions, ce qui aurait pour effet de ne permettre que la modification des `Categorie` qui sont *déjà liées* à l'`Article`.

Assez parlé, testons dès maintenant le résultat. Pour cela, actualisez la page d'ajout d'un article. Ah mince, le mot « `Categorie` » est bien inscrit, mais il n'y a rien en dessous. 😞 Ce n'est pas un bug, c'est bien voulu par Symfony2. En effet, comme l'entité `Article` lié au formulaire de base n'a pas encore de catégories, le champ `collection` n'a encore rien à afficher ! Et si on veut créer des catégories, il ne peut pas savoir à l'avance combien on veut en créer : 1, 2, 3 ou plus ?

La solution, sachant qu'on doit pouvoir ajouter à l'infini, et même supprimer, est d'utiliser du JavaScript. OK, cela ne nous fait pas peur !

D'abord, affichez la source de la page et regardez l'étrange balise `<div>` que Symfony2 a rajoutée en dessous du label `Categorie` :

Code : HTML

```
<div id="sdz_blogbundle_articletype_categories" data-prototype="<div><label class="required" for="sdz_blogbundle_articletype_categories_name_nom">_name_label_</label></div>
<label for="sdz_blogbundle_articletype_categories_name_nom" class="required">Nom</label><input type="text" id="sdz_blogbundle_articletype_categories_name_nom" name="sdz_blogbundle_articletype[categories][name][nom]" required="required"/>
</div></div></div>">
```

Notez surtout l'attribut `data-prototype`. C'est en fait un attribut (au nom arbitraire) rajouté par Symfony2 et qui contient ce à quoi doit ressembler le code HTML pour ajouter un formulaire `CategorieType`. Voici son contenu sans les entités HTML :

Code : HTML

```
<div>
  <label class="required">_name_label_</label>
  <div id="sdz_blogbundle_articletype_categories_name_">
    <div>
      <label
        for="sdz_blogbundle_articletype_categories_name_nom"
        class="required">Nom</label>
      <input type="text"
            id="sdz_blogbundle_articletype_categories_name_nom"
            name="sdz_blogbundle_articletype[categories][name][nom]"
            required="required" />
    </div>
  </div>
</div>
```

Vous voyez qu'il contient les balises `<label>` et `<input>`, tout ce qu'il faut pour créer le champ `nom` compris dans `CategorieType`, en fait. Si ce formulaire avait d'autres champs en plus de « `nom` », ceux-ci apparaîtraient ici également.

Du coup, on le remercie, car grâce à ce template ajouter des champs en JavaScript est un jeu d'enfant. Je vous propose de faire un petit script JavaScript dont le but est :

- D'ajouter un bouton `Ajouter` qui permet d'ajouter à l'infini ce sous-formulaire `CategorieType` contenu dans l'attribut `data-prototype` ;

- D'ajouter pour chaque sous-formulaire, un bouton Supprimer permettant de supprimer la catégorie associée.

Voici ce que je vous ai préparé, un petit script qui emploie la bibliothèque jQuery, mettez-le pour l'instant directement dans la vue du formulaire :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/formulaire.html.twig #-}

{# Le formulaire reste inchangé #}
<div class="well">
    <form method="post" {{ form_enctype(form) }}>
        {{ form_widget(form) }}
        <input type="submit" class="btn btn-primary" />
    </form>
</div>

{# On charge la bibliothèque jQuery. Ici, je la prends depuis le
site jquery.com,
mais si vous l'avez en local, changez simplement l'adresse. #}
<script src="http://code.jquery.com/jquery-1.8.2.min.js"></script>

{# Voici le script en question : #}
<script type="text/javascript">
$(document).ready(function() {
    // On récupère la balise <div> en question qui contient
    l'attribut « data-prototype » qui nous intéresse.
    var $container = $('#sdz_blogbundle_articletype_categories');

    // On ajoute un lien pour ajouter une nouvelle catégorie
    var $lienAjout = $('Ajouter une catégorie</a>'\);
    \$container.append\(\$lienAjout\);

    // On ajoute un nouveau champ à chaque clic sur le lien d'ajout.
    \$lienAjout.click\(function\(e\) {
        ajouterCategorie\(\$container\);
        e.preventDefault\(\); // évite qu'un # apparaisse dans l'URL
        return false;
    }\);

    // On définit un compteur unique pour nommer les champs qu'on va
    ajouter dynamiquement
    var index = \$container.find\(':input'\).length;

    // On ajoute un premier champ directement s'il n'en existe pas
    déjà un \(cas d'un nouvel article par exemple\).
    if \(index == 0\) {
        ajouterCategorie\(\$container\);
    } else {
        // Pour chaque catégorie déjà existante, on ajoute un lien de
        suppression
        \$container.children\('div'\).each\(function\(\) {
            ajouterLienSuppression\(\$\(this\)\);
        }\);
    }

    // La fonction qui ajoute un formulaire Categorie
    function ajouterCategorie\(\$container\) {
        // Dans le contenu de l'attribut « data-prototype », on
        remplace :
        // - le texte "\_\_name\_\_label\_\_" qu'il contient par le label du
        champ
        // - le texte "\_\_name\_\_" qu'il contient par le numéro du champ
        var \$prototype = \$\(\$container.attr\('data-
        prototype'\)\).replace\(/\_\_name\_\_label\_\_/g, 'Catégorie n°' + \(index+1\)\)
        .replace\(/\_\_name\_\_/g, index\)\);
    }
}
```

```
// On ajoute au prototype un lien pour pouvoir supprimer la
// catégorie
ajouterLienSuppression($prototype);

// On ajoute le prototype modifié à la fin de la balise <div>
$container.append($prototype);

// Enfin, on incrémente le compteur pour que le prochain ajout
se fasse avec un autre numéro
index++;
}

// La fonction qui ajoute un lien de suppression d'une catégorie
function ajouterLienSuppression($prototype) {
    // Création du lien
    $lienSuppression = $(<a href="#" class="btn btn-
danger">Supprimer</a>');

    // Ajout du lien
    $prototype.append($lienSuppression);

    // Ajout du listener sur le clic du lien
    $lienSuppression.click(function(e) {
        $prototype.remove();
        e.preventDefault(); // évite qu'un # apparaisse dans l'URL
        return false;
    });
}
});
</script>
```

Appuyez sur F5 sur la page d'ajout et admirez le résultat (voir figure suivante). Voilà qui est mieux !

The screenshot shows a user interface for managing categories. At the top left is a button labeled "Ajouter une catégorie". Below it, there are two sections, each containing a label ("Catégorie n°1" and "Catégorie n°2"), a text input field for "Nom", and a red "Supprimer" button. At the bottom right of the interface is a blue "Valider" button.

Le formulaire multiple est opérationnel et dynamique

Et voilà, votre formulaire est maintenant opérationnel ! Vous pouvez vous amuser à créer des articles contenant plein de nouvelles catégories en même temps.

Un type de champ très utile : entity

Je vous ai prévenu que ce qu'on vient de faire sur l'attribut `categories` était particulier : sur le formulaire d'ajout d'un article nous pouvons **créer** des nouvelles catégories et non **sélectionner** des catégories déjà existantes. Ce paragraphe n'a rien à voir avec l'imbrication de formulaire, mais je me dois de vous en parler maintenant pour que vous compreniez bien la différence entre les types de champ `entity` et `collection`.

Le type `entity` est un type assez puissant, vous allez le voir très vite. Nous allons l'utiliser à la place du type `collection` qu'on vient de mettre en place. Vous connaîtrez ainsi les deux types, libre à vous ensuite d'utiliser celui qui convient le mieux à votre cas.

Le type `entity` permet donc de sélectionner des entités. D'un `<select>` côté formulaire HTML, vous obtenez une ou plusieurs entités côté formulaire Symfony2. Testons-le tout de suite, modifiez le champ `categories` comme suit :

Code : PHP

```
<?php
// src/Szd/BlogBundle/Form/ArticleType.php

// ...

$builder->add('categories', 'entity', array(
    'class'      => 'SdzBlogBundle:Categorie',
    'property'   => 'nom',
    'multiple'   => true
));
```

Rafraîchissez le formulaire et admirez :

The screenshot shows a form field labeled "Categories". Inside the field is a dropdown menu containing four items: "Symfony2", "Doctrine2", "Tutoriel", and "Evènement". Below the dropdown is a blue rectangular button with the text "Valider".

On peut ainsi sélectionner une ou plusieurs catégories

Les options du type de champ

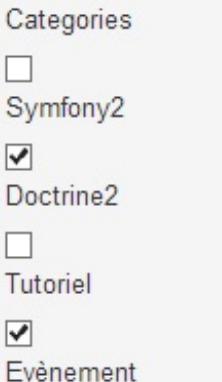
Alors, quelques explications sur les options de ce type de champ :

- L'option `class` définit quel est le type d'entité à sélectionner. Ici, on veut sélectionner des entités `Categorie`, on renseigne donc le raccourci `Doctrine` pour cette entité (ou son namespace complet).
- L'option `property` définit comment afficher les entités dans le `select` du formulaire. En effet, comment afficher une catégorie ? Par son nom ? Son id ? Un mix des deux ? Ce n'est pas à Symfony de le deviner, on lui précise donc grâce à cette option `property`. Ici j'ai renseigné `nom`, c'est donc via leur nom qu'on liste les catégories dans le `select`. Sachez que vous pouvez également renseigner `affichage` (ou autre !) et créer le getter associé (à savoir `getAffichage()`) dans l'entité `Categorie`, ce sera donc le retour de cette méthode qui sera affiché dans le `select`.
- L'option `multiple` définit qu'on parle ici d'une liste de catégories, et non d'une catégorie unique. Cette option est très importante, car, si vous l'oubliez, le formulaire (qui retourne une entité `Categorie`) et votre entité `Article` (qui attend une liste d'entités `Categorie`) ne vont pas s'entendre !

Alors, intéressant, ce type de champ, n'est-ce pas ?

Et encore, ce n'est pas fini. Si la fonctionnalité de ce type (sélectionner une ou plusieurs entités) est unique, le rendu peut avoir quatre formes en fonction des options `multiple` et `expanded` :

Options	<code>Multiple = false</code>	<code>Multiple = true</code>
		<code><select></code> avec plusieurs options sélectionnables

Expanded = false	<select> avec une seule option sélectionnable			
Expanded = true	<radio>		<checkbox>	

Par défaut, les options multiple et expanded sont à false. 😊

L'option querybuilder

Comme vous avez pu le constater, toutes les catégories de la base de données apparaissent dans ce champ. Or parfois ce n'est pas le comportement voulu. Imaginons par exemple un champ où vous souhaitez afficher uniquement les articles *publiés*. Tout est prévu : il faut jouer avec l'option querybuilder.

Cette option porte bien son nom puisqu'elle permet de passer au champ un QueryBuilder, que vous connaissez depuis la partie sur Doctrine. Tout d'abord, créons une méthode dans le repository de l'entité du champ qui retourne le bon QueryBuilder :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

namespace Sdz\BlogBundle\Entity;
use Doctrine\ORM\EntityRepository;

class ArticleRepository extends EntityRepository
{
    public function getSelectList()
    {
        $qb = $this->createQueryBuilder('a')
            ->where('a.publication = 1'); // On filtre sur
        l'attribut publication

        // Et on retourne simplement le QueryBuilder, et non la Query,
        attention
        return $qb;
    }

    // ...
}
```

Et on l'envoie dans l'option querybuilder grâce à une closure dont l'argument est le repository, comme ceci :

Code : PHP

```
<?php
// Dans un XxxType

$builder->add('article', 'entity', array(
    'class'          => 'SdzBlogBundle:Article',
    'property'       => 'titre',
    'querybuilder'   =>
function (\Sdz\BlogBundle\Entity\ArticleRepository $r) {
    return $r->getSelectList();
}
));
```

Si vous n'êtes pas habitués aux closures, voici la méthode pour passer un argument de votre formulaire jusqu'à la méthode du repository, il vous faut utiliser le `use ()` de PHP, comme ceci :

Code : PHP

```
<?php
'querybuilder' => function(ArticleRepository $r) use($uneVariable) {
    return $r->getSelectList($uneVariable);
}
```

Souvenez-vous de cette syntaxe, elle vous servira très certainement un jour. 😊

Aller plus loin avec les formulaires

L'héritage de formulaire

Je souhaiterais vous faire un point sur l'héritage de formulaire. En effet, nos formulaires, représentés par les objets `XxxType`, sont avant tout de simples objets ! On peut donc tout à fait utiliser les mécanismes habituels en POO.

L'utilité dans le cadre des formulaires, c'est de pouvoir construire des formulaires différents, mais ayant la même base. Pour faire simple, je vais prendre l'exemple des formulaires d'ajout et de modification d'un `Article`. Imaginons que le formulaire d'ajout comprenne tous les champs, mais que pour l'édition il soit impossible de modifier la date par exemple.

Comme nous sommes en présence de deux formulaires distincts, on va faire deux `XxxType` distincts : `ArticleType` pour l'ajout, et `ArticleEditType` pour la modification. Seulement, il est hors de question de répéter la définition de tous les champs dans le `ArticleEditType`, tout d'abord c'est long, mais surtout si jamais un champ change, on devra modifier à la fois `ArticleType` et `ArticleEditType`, c'est impensable.

On va donc faire hériter `ArticleEditType` de `ArticleType`. Le processus est le suivant :

1. Copiez-collez le fichier `ArticleType.php` et renommez la copie en `ArticleEditType.php` ;
2. Modifiez le nom de la classe, ainsi que le nom du formulaire dans la méthode `getName()` ;
3. Remplacez la définition manuelle de tous les champs (les `$builder->add()`) par un appel à la méthode parente : `<?php parent::buildForm($builder, $options);`
4. Rajoutez cette ligne à la suite pour supprimer le champ `date` et ainsi ne pas le faire apparaître lors de l'édition : `<?php $builder->remove('date');`
5. Enfin, supprimez la méthode `setDefaultOptions()` qu'il ne sert à rien d'hériter.

Voici ce que cela donne :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Form/ArticleEditType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
```

```

class ArticleEditType extends ArticleType // Ici, on hérite de ArticleType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // On fait appel à la méthode buildForm du parent, qui va ajouter tous les champs à $builder
        parent::buildForm($builder, $options);

        // On supprime celui qu'on ne veut pas dans le formulaire de modification
        $builder->remove('date');
    }

    // On modifie cette méthode car les deux formulaires doivent avoir un nom différent
    public function getName()
    {
        return 'sdz_blogbundle_articleedittype';
    }
}

```

Nous avons utilisé ici des notions de POO en PHP, qui n'ont rien du tout de spécifique à Symfony2. Avant d'être des formulaires, des entités et autres, nos objets sont de simples objets PHP avec lesquels on peut faire tout ce que PHP nous autorise. 😊 Gardez toujours cela en tête.



Maintenant, si vous utilisez le formulaire `ArticleEditType`, vous ne pourrez pas modifier l'attribut `date` de l'entité `Article`. Objectif atteint ! Prenez le temps de tester ce nouveau formulaire depuis l'action `modifierAction()` de notre blog.

À retenir

Plusieurs choses à retenir de cet héritage de formulaire :

- D'une part, si vous avez besoin de plusieurs formulaires : faites plusieurs `XxxType` ! Cela ne mange pas de pain, et vous évitez de faire du code impropre derrière en mettant des conditions hasardeuses. Le raisonnement est simple : si le formulaire que vous voulez afficher à votre internaute est différent (champ en moins, champ en plus), alors côté Symfony2 c'est un tout autre formulaire, qui mérite son propre `XxxType`.
- D'autre part, pensez à bien utiliser la notion PHP d'héritage pour éviter de dupliquer du code. Si faire plusieurs formulaires est une bonne chose, dupliquer les champs à droite et à gauche ne l'est pas. Centralisez donc la définition de vos champs dans un formulaire, et utilisez l'héritage pour le propager aux autres.

Construire un formulaire différemment selon des paramètres

Un autre besoin qui se fait sentir lors de l'élaboration de formulaires un peu plus complexes que notre simple `ArticleType`, c'est la modulation d'un formulaire en fonction de certains paramètres.

Par exemple, on pourrait empêcher de dépublier un article une fois qu'il est publié. Le comportement serait le suivant :

- Si l'article n'est pas encore publié, on peut modifier sa valeur de publication lorsqu'on modifie l'article ;
- Si l'article est déjà publié, on ne peut plus modifier sa valeur de publication lorsqu'on modifie l'article.

C'est un exemple simple, retenez l'idée derrière qui est de construire le formulaire suivant les valeurs de l'objet sous-jacent. Ce n'est pas aussi évident qu'il n'y paraît, car dans la méthode `buildForm()` nous n'avons pas accès aux valeurs de l'objet `Article` qui sert de base au formulaire !

Pour arriver à nos fins, il faut utiliser les évènements de formulaire. Ce sont des évènements que le formulaire déclenche à certains moments de sa construction. Il existe notamment l'évènement `PRE_SET_DATA` qui est déclenché juste avant que les champs ne soient remplis avec les valeurs de l'objet (les valeurs par défaut donc). Cet évènement permet de modifier la structure du formulaire.

Sans plus attendre, voici à quoi ressemble notre nouvelle méthode `buildForm()` :

Code : PHP

```
<?php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
// Ajoutez ces deux use :
use Symfony\Component\Form\FormEvents;
use Symfony\Component\Form\FormEvent;

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
            // ... Ajoutez ici tous les champs stables, mais pas le champ
public
            ;
        $factory = $builder->getFormFactory();

        // On ajoute une fonction qui va écouter l'évènement
        PRE_SET_DATA
            $builder->addEventListener(
                FormEvents::PRE_SET_DATA, // Ici, on définit l'évènement qui
nous intéresse
                function(FormEvent $event) use ($factory) { // Ici, on définit
une fonction qui sera exécutée lors de l'évènement
                    $article = $event->getData();
                    // Cette condition est importante, on en reparle plus loin
                    if (null === $article) {
                        return; // On sort de la fonction lorsque $article vaut
null
                    }
                    // Si l'article n'est pas encore publié, on ajoute le champ
public
                    if (false === $article->getPublication()) {
                        $event->getForm()->add(
                            $factory->createNamed('publication', 'checkbox', null,
array('required' => false))
                        );
                    } else { // Sinon, on le supprime
                        $event->getForm()->remove('publication');
                    }
                }
            );
        // ...
    }
}
```

Il y a beaucoup de syntaxe dans ce code, mais il est au fond abordable, et vous montre les possibilités qu'offrent les évènements de formulaire.

La fonction qui est exécutée par l'évènement prend en argument l'évènement lui-même, la variable `$event`. Depuis cet objet évènement, vous pouvez récupérer d'une part l'objet sous-jacent, via `$event->getData()`, et d'autre part le formulaire, via

```
$event->getForm().
```

Récupérer l'Article nous permet d'utiliser les valeurs qu'il contient, chose qu'on ne peut pas faire d'habitude dans la méthode buildForm(), qui, elle, est exécutée une fois pour toutes, indépendamment de l'objet sous-jacent. Pour mieux visualiser cette unique instance du XxxType, pensez à un champ de type collection, rappelez-vous sa définition :

Code : PHP

```
<?php
$builder->add('categories', 'collection', array('type' => new
CategorieType()));
```

Avec ce code, on ne crée qu'un seul objet CategorieType, or celui-ci sera utilisé pour ajouter plusieurs catégories. Il est donc normal de ne pas avoir accès à l'objet \$categorie lors de la construction du formulaire, autrement dit la construction de l'objet CategorieType. C'est pour cela qu'il faut utiliser l'évènement PRE_SET_DATA, qui, lui, est déclenché à chaque fois que le formulaire remplit les valeurs de ses champs par les valeurs de l'objet \$categorie.

 Je reviens sur la condition if (null == \$article) dans la fonction. En fait, à la première création du formulaire, celui-ci exécute sa méthode setData() avec null en argument. Cette occurrence de l'évènement PRE_SET_DATA ne nous intéresse pas, d'où la condition pour sortir de la fonction lorsque \$event->getData() vaut null. Ensuite, lorsque le formulaire récupère l'objet (\$article dans notre cas) sur lequel se construire, il réexécute sa méthode setData() avec l'objet en argument. C'est cette occurrence-là qui nous intéresse.

Sachez qu'il est également possible d'ajouter non pas une simple fonction à exécuter lors de l'évènement, mais un service ! Tout cela et bien plus encore est décrit dans [la documentation des évènements de formulaire](#). N'hésitez pas à vous documenter dessus, car c'est cette méthode des évènements qui permet également la création des fameuses *combo box* : deux champs <select> dont le deuxième (par exemple ville) dépend de la valeur du premier (par exemple pays).

Le type de champ File pour envoyer des fichiers

Dans cette partie, nous allons apprendre à envoyer un fichier via le type File, ainsi qu'à le persister via les évènements Doctrine (j'espère que vous ne les avez pas déjà oubliés !).

Le type de champ File

Un champ File ne retourne pas du texte, mais une instance de la classe `UploadedFile`. Pour cette raison, il faut utiliser un attribut à part dans l'entité sous-jacente au formulaire, ici `Image`.

Préparer l'objet sous-jacent

Ouvrez donc l'entité `Image` et ajoutez l'attribut `$file` suivant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Image.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ImageRepository")
 */
class Image
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @Assert\NotBlank()
     */
    private $id;

    /**
     * @ORM\Column(name="name", type="string", length=255)
     */
    private $name;

    /**
     * @ORM\Column(name="type", type="string", length=255)
     */
    private $type;

    /**
     * @ORM\Column(name="size", type="integer")
     */
    private $size;

    /**
     * @ORM\Column(name="path", type="string", length=255)
     */
    private $path;

    /**
     * @ORM\Column(name="file", type="blob")
     */
    private $file;
}
```

```


    /**
 * @ORM\GeneratedValue(strategy="AUTO")
 */
private $id;

/**
 * @ORM\Column(name="url", type="string", length=255)
 */
private $url;

/**
 * @ORM\Column(name="alt", type="string", length=255)
 */
private $alt;

private $file;

// ...
}


```

Notez bien que je n'ai pas mis d'annotation pour Doctrine : ce n'est pas cet attribut `$file` que nous allons persister par la suite, on ne met donc pas d'annotation. Par contre, c'est bien cet attribut qui servira pour le formulaire, et non les autres.

Adapter le formulaire

Passons maintenant au formulaire. Nous avions construit un champ de formulaire sur l'attribut `$url`, dans lequel l'utilisateur devait mettre directement l'URL internet de son image. Maintenant on veut lui permettre d'envoyer un fichier depuis son ordinateur.

On va donc supprimer le champ sur `$url` (et sur `$alt`, on va pouvoir le générer dynamiquement) et en créer un nouveau sur `$file` :

Code : PHP

```


<?php
// src/Sdz/BlogBundle/Form/ImageType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ImageType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
            ->add('file', 'file')
        ;
    }

    // ...
}


```

Le rendu de votre formulaire est déjà bon. Essayez de vous rendre sur la page d'ajout, vous allez voir le champ d'upload de la figure suivante.

Auteur

Image

File

Choisissez un fichier Aucun...oisi

Catégories

Ajouter une catégorie

Champ pour envoyer un fichier



Lorsque vous utilisez des formulaires avec des envois de fichiers, n'oubliez jamais de préciser l'`enctype` dans la balise HTML du formulaire, comme ceci : `<form method="post" {{ form_enctype(form) }}>`.

Bon, par contre évidemment le formulaire n'est pas opérationnel. La sauvegarde du fichier envoyé ne va pas se faire toute seul !

Manipuler le fichier envoyé

Une fois le formulaire soumis, il faut bien évidemment s'occuper du fichier envoyé. L'objet `UploadedFile` que le formulaire nous renvoie simplifie grandement les choses, grâce à sa méthode `move()`. Créons une méthode `upload()` dans notre objet `Image` pour s'occuper de tout cela :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Image.php

public function upload()
{
    // Si jamais il n'y a pas de fichier (champ facultatif)
    if (null === $this->file) {
        return;
    }

    // On garde le nom original du fichier de l'internaute
    $name = $this->file->getClientOriginalName();

    // On déplace le fichier envoyé dans le répertoire de notre
    // choix
    $this->file->move($this->getUploadRootDir(), $name);

    // On sauvegarde le nom de fichier dans notre attribut $url
    $this->url = $name;

    // On crée également le futur attribut alt de notre balise
    <img>
        $this->alt = $name;
    </img>

    public function getUploadDir()
    {
        // On retourne le chemin relatif vers l'image pour un
        // navigateur
        return 'uploads/img';
    }

    protected function getUploadRootDir()
    {
        // On retourne le chemin relatif vers l'image pour notre code
        // PHP
        return __DIR__.'../../../../web/'.$this->getUploadDir();
    }
}
```

```
}
```

Plusieurs choses dans ce code.

D'une part, on a défini le répertoire dans lequel stocker nos images. J'ai mis ici `uploads/img`, ce répertoire est relatif au répertoire web, vous pouvez tout à fait le personnaliser. La méthode `getUploadDir()` retourne ce chemin relatif, à utiliser dans vos vues car les navigateurs sont relatifs à notre répertoire web. La méthode `getUploadRootDir()`, quant à elle, retourne le chemin vers le même fichier, mais en absolu. Vous le savez `__DIR__` représente le répertoire absolu du fichier courant, ici notre entité, du coup pour atteindre le répertoire web, il faut remonter pas mal de dossiers, comme vous pouvez le voir.

D'autre part, la méthode `upload()` s'occupe concrètement de notre fichier. Elle fait l'équivalent du `move_uploaded_file()` que vous pouviez utiliser en PHP pur. Ici j'ai choisi pour l'instant de garder le nom du fichier tel qu'il était sur le PC du visiteur, ce n'est évidemment pas optimal, car si deux fichiers du même nom sont envoyés, le second écrasera le premier !

Enfin, d'un point de vue persistance de notre entité `Image` dans la base de données, la méthode `upload()` s'occupe également de renseigner les deux attributs persistés, `$url` et `$salt`. En effet, l'attribut `$file`, qui est le seul rempli par le formulaire, n'est pas du tout persisté.

Bien entendu, cette méthode ne s'exécute pas toute seule, il faut l'exécuter à la main depuis le contrôleur. Rajoutez donc la ligne 8 du code suivant dans la méthode `ajouterAction()` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...

if ($form->isValid()) {
    // Ici : On traite manuellement le fichier uploadé
    $article->getImage()->upload();

    // Puis, le reste de la méthode, qu'on avait déjà fait
    $em = $this->getDoctrine()->getManager();
    $em->persist($article);
    $em->flush();

    // Enfin, éventuelle redirection vers la page du nouvel article
    // créé
    return $this->redirect($this->generateUrl('sdzblog_voir',
        array('id' => $article->getId())));
}

// ...
```



Il est impératif d'exécuter la méthode `upload()` avant de persister l'entité, car sinon les attributs `$url` et `$salt` ne seront pas définis à l'exécution du `flush`, et cela créerait une erreur (ils ne peuvent pas être `null` dans la base de données).

Si vous commencez à bien penser « découplage », ce que nous venons de faire ne devrait pas vous plaire. Le contrôleur ne devrait pas avoir à agir juste parce que nous avons un peu modifié le comportement de l'entité `Image`. Et imaginez qu'un jour nous oubliions d'exécuter manuellement cette méthode `upload()` ! Bref, vous l'aurez compris, il faut ici réutiliser les événements Doctrine2 pour automatiser tout cela.

Automatiser le traitement grâce aux évènements

La manipulation du champ de type `File` que nous venons de faire est bonne, mais son implémentation est juste un peu maladroite. Il faut automatiser cela grâce aux événements Doctrine. Mais ce n'est pas que de l'esthétisme, c'est impératif pour gérer tous les cas... comme la suppression par exemple !

On va également en profiter pour modifier le nom donné au fichier qu'on déplace dans notre répertoire `web/uploads/img`. Le fichier va prendre comme nom l'`id` de l'entité, suffixé de son extension évidemment.

Quels évènements utiliser ?

C'est une question qu'il faut toujours se poser consciencieusement, car le comportement peut changer du tout au tout suivant les évènements choisis. Dans notre cas, il y a en réalité quatre actions différentes à exécuter :

- Avant l'enregistrement effectif dans la base de données : il faut remplir les attributs `$url` et `$alt` avec les bonnes valeurs suivant le fichier envoyé. On doit impérativement le faire avant l'enregistrement, pour qu'ils puissent être enregistrés eux-mêmes en base de données. Pour cette action, il faut utiliser les évènements :
 - `PrePersist`
 - `PreUpdate`
- Juste après l'enregistrement : il faut déplacer effectivement le fichier envoyé. On ne le fait pas avant, car l'enregistrement peut échouer. En cas d'échec de l'enregistrement de l'entité en base de données, il ne faudrait pas se retrouver avec un fichier orphelin sur notre disque. On attend donc que l'enregistrement se fasse effectivement avant de déplacer le fichier. Pour cette action, il faut utiliser les évènements :
 - `PostPersist`
 - `PostUpdate`
- Juste avant la suppression : il faut sauvegarder le nom du fichier dans un attribut non persisté, `$filename`. En effet, le nom du fichier dépendant de l'`id`, on n'y aura plus accès en `PostRemove`, on est donc obligé de le sauvegarder en `PreRemove` : peu pratique mais obligatoire. Pour cette action, il faut utiliser l'évènement :
 - `PreRemove`
- Juste après la suppression : il faut supprimer le fichier qui était associé à l'entité. Encore une fois, on ne le fait pas avant la suppression, car si l'entité n'est au final pas supprimée, on aurait alors une entité sans fichier. Pour cette action, il faut utiliser l'évènement :
 - `PostRemove`

Implémenter les méthodes des évènements

La méthode est la suivante :

- On éclate l'ancien code de la méthode `upload()` dans les méthodes :
 - `preUpload()` : pour ce qui est de la génération des attributs `$url` et `$alt` ;
 - `upload()` : pour le déplacement effectif du fichier.
- On ajoute une méthode `preRemoveUpload()` qui sauvegarde le nom du fichier qui dépend de l'`id` de l'entité.
- On ajoute une méthode `removeUpload()` qui supprime effectivement le fichier grâce au nom enregistré.

N'oubliez pas de rajouter un attribut (ici j'ai mis `$filename`) pour la sauvegarde du nom du fichier. Au final, voici ce que cela donne :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Image.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
// N'oubliez pas ce use :
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * 
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ImageRepository")
 * @ORM\HasLifecycleCallbacks
 */

```

```
class Image
{
    private $file;

    // On ajoute cet attribut pour y stocker le nom du fichier
    // temporairement
    private $tempFilename;

    // On modifie le setter de File, pour prendre en compte l'upload
    // d'un fichier lorsqu'il en existe déjà un autre
    public function setFile(UploadedFile $file)
    {
        $this->file = $file;

        // On vérifie si on avait déjà un fichier pour cette entité
        if (null !== $this->url) {
            // On sauvegarde l'extension du fichier pour le supprimer
            // plus tard
            $this->tempFilename = $this->url;

            // On réinitialise les valeurs des attributs url et alt
            $this->url = null;
            $this->alt = null;
        }
    }

    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        // Si jamais il n'y a pas de fichier (champ facultatif)
        if (null === $this->file) {
            return;
        }

        // Le nom du fichier est son id, on doit juste stocker
        // également son extension
        // Pour faire propre, on devrait renommer cet attribut en «
        // extension », plutôt que « url »
        $this->url = $this->file->guessExtension();

        // Et on génère l'attribut alt de la balise <img>, à la valeur
        // du nom du fichier sur le PC de l'internaute
        $this->alt = $this->file->getClientOriginalName();
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        // Si jamais il n'y a pas de fichier (champ facultatif)
        if (null === $this->file) {
            return;
        }

        // Si on avait un ancien fichier, on le supprime
        if (null !== $this->tempFilename) {
            $oldFile = $this->getUploadRootDir().'/'.$this->id.'.'.$this-
>tempFilename;
            if (file_exists($oldFile)) {
                unlink($oldFile);
            }
        }

        // On déplace le fichier envoyé dans le répertoire de notre
        // choix
    }
}
```

```

    $this->file->move(
        $this->getUploadRootDir(), // Le répertoire de destination
        $this->id.'.'.$this->url // Le nom du fichier à créer, ici
    << id.extension >>
    );
}

/**
* @ORM\PreRemove()
*/
public function preRemoveUpload()
{
    // On sauvegarde temporairement le nom du fichier, car il
    dépend de l'id
    $this->tempFilename = $this->getUploadRootDir().'/'.$this-
>id.'.'.$this->url;
}

/**
* @ORM\PostRemove()
*/
public function removeUpload()
{
    // En PostRemove, on n'a pas accès à l'id, on utilise notre nom
    sauvegardé
    if (file_exists($this->tempFilename)) {
        // On supprime le fichier
        unlink($this->tempFilename);
    }
}

public function getUploadDir()
{
    // On retourne le chemin relatif vers l'image pour un
    navigateur
    return 'uploads/img';
}

protected function getUploadRootDir()
{
    // On retourne le chemin relatif vers l'image pour notre code
    PHP
    return __DIR__.'../../../../web/'.$this->getUploadDir();
}

// ...
}

```

Et voilà, votre upload est maintenant totalement opérationnel.



Bien sûr, vous devez supprimer l'appel à `$article->getImage()->upload()` qu'on avait mis à la main dans le contrôleur. Cette ligne n'est plus utile maintenant que tout est fait automatiquement grâce aux événements !

Vous pouvez vous amuser avec votre système d'upload. Créez des articles avec des images jointes, vous verrez automatiquement les fichiers apparaître dans `web/uploads/img`. Supprimez un article : l'image jointe sera automatiquement supprimée du répertoire.



Pour que l'entité `Image` liée à un article soit supprimée lorsque vous supprimez l'entité `Article`, assurez-vous que l'action `remove` soit en cascade. Pour cela, votre annotation sur l'attribut `$image` dans votre entité `Article` devrait ressembler à ceci :

`@ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image", cascade={"persist", "remove"}).`



Attention à ne pas laisser la possibilité à vos visiteurs d'envoyer n'importe quel type de fichier sur votre site internet ! Il est impératif d'ajouter une règle de validation `@Assert\File` pour limiter les types de fichiers, et ne pas laisser une faille de sécurité béante. On aborde les règles de validation dans le prochain chapitre.

Vous devez également modifier la vue `article.html.twig` qui affiche les images. Nous avions utilisé `{ { image.url } }`, mais ce n'est plus bon puisque l'on ne stocke plus que l'extension du fichier dans l'attribut `$url`. Il faudrait donc mettre le code suivant :

Code : HTML & Django

```

```

En fait, comme vous pouvez le voir, c'est assez long à écrire dans la vue. Il est donc intéressant d'ajouter une méthode qui fait tout cela dans l'entité, par exemple `getWebPath()` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Image.php

public function getWebPath()
{
    return $this->getUploadDir() . '/' . $this->getId() . '.' . $this-
>getUrl();
}
```

Et du coup, dans la vue, il ne reste plus que :

Code : HTML & Django

```

```

Application : les formulaires de notre blog

Théorie

Nous avons déjà généré presque tous les formulaires utiles pour notre blog, mais nous n'avons pas entièrement adapté les actions du contrôleur pour les rendre pleinement opérationnelles.

Je vous invite donc à reprendre tout notre contrôleur, et à le modifier de telle sorte que toutes ses actions soient entièrement fonctionnelles, vous avez toutes les clés en main maintenant ! Je pense notamment aux actions de modification et de suppression, que nous n'avons pas déjà faites dans ce chapitre. Au boulot ! Essayez d'implémenter vous-mêmes la gestion du formulaire dans les actions correspondantes. Ensuite seulement, lisez la suite de ce paragraphe pour avoir la solution.

Pratique

Je vous remets déjà tous les formulaires pour être sûr qu'on parle de la même chose.

`ArticleType`

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Form/ArticleType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
use Symfony\Component\Form\FormEvents;
use Symfony\Component\Form\FormEvent;

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('date', 'date')
            ->add('titre', 'text')
            ->add('contenu', 'textarea')
            ->add('auteur', 'text')
            ->add('image', new ImageType())
            /*
            * Rappel :
            ** - 1er argument : nom du champ, ici « categories » car c'est le
            nom de l'attribut
            ** - 2e argument : type du champ, ici « collection » qui est une
            liste de quelque chose
            ** - 3e argument : tableau d'options du champ
            */
            /*->add('categories', 'collection', array('type' => new
            CategorieType(),
            'allow_add' => true,
            'allow_delete' => true))
            ->add('categories', 'entity', array(
            'class' => 'SdzBlogBundle:Categorie',
            'property' => 'nom',
            'multiple' => true,
            'expanded' => false
            ))
        ;
        // On récupère la factory (usine)
        $factory = $builder->getFormFactory();

        // On ajoute une fonction qui va écouter l'évènement
PRE_SET_DATA
        $builder->addEventListener(
            FormEvents::PRE_SET_DATA, // Ici, on définit l'évènement qui
            nous intéresse
            function(FormEvent $event) use ($factory) { // Ici, on définit
            une fonction qui sera exécutée lors de l'évènement
                $article = $event->getData();
                // Cette condition est importante, on en reparle plus loin
                if (null === $article) {
                    return; // On sort de la fonction lorsque $article vaut
                null
                }
                // Si l'article n'est pas encore publié, on ajoute le champ
                publication
                if (false === $article->getPublication()) {
                    $event->getForm()->add(
                        $factory->createNamed('publication', 'checkbox', null,
                        array('required' => false))
                    );
                } else { // Sinon, on le supprime
                    $event->getForm()->remove('publication');
                }
            }
        );
    }
}

```

```

        }
    );
}

public function setDefaultOptions(OptionsResolverInterface
$resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Sdz\BlogBundle\Entity\Article'
    ));
}

public function getName()
{
    return 'sdz_blogbundle_articletype';
}
}

```

ArticleEditType

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Form/ArticleEditType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ArticleEditType extends ArticleType // Ici, on hérite de ArticleType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        // On fait appel à la méthode buildForm du parent, qui va ajouter tous les champs à $builder
        parent::buildForm($builder, $options);

        // On supprime celui qu'on ne veut pas dans le formulaire de modification
        $builder->remove('date');
    }

    // On modifie cette méthode, car les deux formulaires doivent avoir un nom différent
    public function getName()
    {
        return 'sdz_blogbundle_articleedittype';
    }
}

```

ImageType

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Form/ImageType.php

```

```

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ImageType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('file', 'file')
        ;
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Sdz\BlogBundle\Entity\Image'
        ));
    }

    public function getName()
    {
        return 'sdz_blogbundle_imagetype';
    }
}

```

L'action « ajouter » du contrôleur

On a déjà fait cette action, je vous la remets ici comme référence :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function ajouterAction()
{
    $article = new Article;

    // On crée le formulaire grâce à l'ArticleType
    $form = $this->createForm(new ArticleType(), $article);

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

    // On vérifie qu'elle est de type POST
    if ($request->getMethod() == 'POST') {
        // On fait le lien Requête <-> Formulaire
        $form->bind($request);

        // On vérifie que les valeurs entrées sont correctes
        // (Nous verrons la validation des objets en détail dans le
        // prochain chapitre)
        if ($form->isValid()) {
            // On enregistre notre objet $article dans la base de
            données
            $em = $this->getDoctrine()->getManager();
            $em->persist($article);
            $em->flush();
        }
    }
}

```

```

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

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

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

return $this->render('SdzBlogBundle:Blog:ajouter.html.twig',
array(
    'form' => $form->createView(),
));
}

```

L'action « modifier » du contrôleur

Voici l'une des actions que vous deviez faire tout seuls. Ici pas de piège, il fallait juste penser à bien utiliser ArticleEditType et non ArticleType, car on est en mode édition.

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function modifierAction(Article $article)
{
    // On utiliser le ArticleEditType
    $form = $this->createForm(new ArticleEditType(), $article);

    $request = $this->getRequest();

    if ($request->getMethod() == 'POST') {
        $form->bind($request);

        if ($form->isValid()) {
            // On enregistre l'article
            $em = $this->getDoctrine()->getManager();
            $em->persist($article);
            $em->flush();

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

            return $this->redirect($this->generateUrl('sdzblog_voir',
array('id' => $article->getId())));
        }
    }

    return $this->render('SdzBlogBundle:Blog:modifier.html.twig',
array(
    'form'      => $form->createView(),
    'article'   => $article
));
}

```

L'action « supprimer » du contrôleur

Enfin, voici l'action pour supprimer un article. On la protège derrière un formulaire presque vide. Je dis « presque », car le formulaire va automatiquement contenir un champ CSRF, c'est justement ce que nous recherchons en l'utilisant, pour éviter qu'une faille permette de faire supprimer un article. Vous trouverez plus d'informations sur [la faille CSRF sur Wikipédia](#).

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

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

    $request = $this->getRequest();
    if ($request->getMethod() == 'POST') {
        $form->bind($request);

        if ($form->isValid()) {
            // On supprime l'article
            $em = $this->getDoctrine()->getManager();
            $em->remove($article);
            $em->flush();

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

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

    // Si la requête est en GET, on affiche une page de confirmation avant de supprimer
    return $this->render('SdzBlogBundle:Blog:supprimer.html.twig',
array(
    'article' => $article,
    'form'     => $form->createView()
));
}
```

Je vous invite par la même occasion à faire la vue `supprimer.html.twig`. Voici ce que j'obtiens de mon côté :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/supprimer.html.twig #-}

{%- extends "SdzBlogBundle::layout.html.twig" %}

{%- block title %}
Supprimer un article - {{ parent() }}
{%- endblock %}

{%- block sdzblog_body %}
```

```

<h2>Supprimer un article</h2>

<p>
    Etes-vous certain de vouloir supprimer l'article "{{ article.titre }}" ?
</p>

    {# On met l'id de l'article dans la route de l'action du formulaire #}
    <form action="{{ path('sdzblog_supprimer', {'id': article.id}) }}" method="post">
        <a href="{{ path('sdzblog_voir', {'id': article.id}) }}" class="btn">
            <i class="icon-chevron-left"></i>
            Retour à l'article
        </a>
        <input type="submit" value="Supprimer" class="btn btn-danger" />
        {{ form_rest(form) }}
    </form>

{%
    endblock %
}

```

Le rendu est celui de la figure suivante.

Blog

Supprimer un article

Confirmation de suppression

Etes-vous certain de vouloir supprimer l'article "Mon weekend à Bali !" ?

[Retour à l'article](#) [Supprimer](#)

Pour conclure

Ce chapitre se termine ici. Son contenu est très imposant mais assez cohérent. Dans tous les cas, et plus encore pour ce chapitre, vous devez absolument vous entraîner en parallèle de votre lecture, pour bien assimiler et être sûrs de bien comprendre toutes les notions.

Mais bien entendu, vous ne pouvez pas vous arrêter en si bon chemin. Maintenant que vos formulaires sont opérationnels, il faut bien vérifier un peu ce que vos visiteurs vont y mettre comme données ! C'est l'objectif du prochain chapitre, qui traite de la validation des données, justement. Il vient compléter le chapitre actuel, continuez donc la lecture !

En résumé

- Un formulaire se construit sur un objet existant, et son objectif est d'hydrater cet objet.
- Un formulaire se construit grâce à un `FormBuilder`, et dans un fichier `XxxType` indépendant.
- En développement, le rendu d'un formulaire se fait en une seule ligne grâce à la méthode `{ { form_widget(form) } }`.
- Il est possible d'imbriquer les formulaires grâce aux `XxxType`.
- Le type de champ `collection` affiche une liste de champs d'un certain type.
- Le type de champ `entity` retourne une ou plusieurs entités.
- Il est possible d'utiliser le mécanisme d'héritage pour créer des formulaires différents mais ayant la même base.
- Le type de champ `File` permet l'upload de fichier, et se couple aux entités grâce aux évènements Doctrine.

Validez vos données

Au chapitre précédent nous avons vu comment créer des formulaires avec Symfony2. Mais qui dit formulaire dit vérification des données rentrées ! Symfony2 contient un composant `Validator` qui, comme son nom l'indique, s'occupe de gérer tout cela. Attaquons-le donc !

Pourquoi valider des données ?

Never trust user input

Ce chapitre introduit la validation des objets avec le composant `Validator` de Symfony2. En effet, c'est normalement un des premiers réflexes à avoir lorsque l'on demande à l'utilisateur de remplir des informations : vérifier ce qu'il a rempli ! Il faut toujours considérer que soit il ne sait pas remplir un formulaire, soit c'est un petit malin qui essaie de trouver la faille. Bref, ne jamais faire confiance à ce que l'utilisateur vous donne (« *never trust user input* » en anglais).

La validation et les formulaires sont bien sûr liés, dans le sens où les formulaires ont besoin de la validation. Mais l'inverse n'est pas vrai ! Dans Symfony2, le `validator` est un service indépendant et n'a nul besoin d'un formulaire pour exister. Ayez-le en tête, avec le `validator`, on peut valider n'importe quel objet, entité ou non, le tout sans avoir besoin de formulaire.

L'intérêt de la validation

L'objectif de ce chapitre est donc d'apprendre à définir qu'une entité est valide ou pas. Plus concrètement, il nous faudra établir des règles précises pour dire que tel attribut (le nom d'utilisateur par exemple) doit faire 3 caractères minimum, que tel autre attribut (l'âge par exemple) doit être compris entre 7 et 77 ans, etc. En vérifiant les données avant de les enregistrer en base de données, on est certain d'avoir une base de données cohérente, en laquelle on peut avoir confiance !

La théorie de la validation

La théorie, très simple, est la suivante. On définit des règles de validation que l'on va rattacher à une classe. Puis on fait appel à un service extérieur pour venir lire un objet (instance de ladite classe) et ses règles, et définir si oui ou non l'objet en question respecte ces règles. Simple et logique !

Définir les règles de validation

Les différentes formes de règles

Pour définir ces règles de validation, ou contraintes, il existe deux moyens :

1. Le premier est d'utiliser les annotations, vous les connaissez maintenant. Leur avantage est d'être situées au sein même de l'entité, et juste à côté des annotations du *mapping* Doctrine2 si vous les utilisez également pour votre *mapping*.
2. Le deuxième est d'utiliser le YAML, XML ou PHP. Vous placez donc vos règles de validation hors de l'entité, dans un fichier séparé.

Les deux moyens sont parfaitement équivalents en termes de fonctionnalités. Le choix se fait donc selon vos préférences. Dans la suite du cours, j'utiliserai les annotations, car je trouve extrêmement pratique de centraliser règles de validation et *mapping* Doctrine au même endroit. Facile à lire et à modifier. 😊

Définir les règles de validation

Préparation

Nous allons prendre l'exemple de notre entité `Article` pour construire nos règles. La première étape consiste à déterminer les règles que nous voulons avec des mots, comme ceci :

- La date doit être une date valide ;
- Le titre doit faire au moins 10 caractères de long ;
- Le contenu ne doit pas être vide ;
- L'auteur doit faire au moins 2 caractères de long ;
- L'image liée doit être valide selon les règles attachées à l'objet `Image`.

À partir de cela, nous pourrons convertir ces mots en annotations.

Annotations

Pour définir les règles de validation, nous allons donc utiliser les annotations. La première chose à savoir est le namespace des annotations à utiliser. Souvenez-vous, pour le *mapping* Doctrine c'était `@ORM`, ici nous allons utiliser `@Assert`, donc le namespace complet est le suivant :

Code : PHP

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

Ce `use` est à rajouter au début de l'objet que l'on va valider, notre entité `Article` en l'occurrence. En réalité, vous pouvez définir l'alias à autre chose qu'`Assert`. Mais c'est une convention qui s'est installée, donc autant la suivre pour avoir un code plus facilement lisible pour les autres développeurs.

Ensuite, il ne reste plus qu'à ajouter les annotations pour traduire les règles que l'on vient de lister. Sans plus attendre, voici donc la syntaxe à respecter. Exemple avec notre objet `Article`:

Code : PHP

```
<?php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
// N'oubliez pas de rajouter ce « use », il définit le namespace pour
// les annotations de validation
use Symfony\Component\Validator\Constraints as Assert;

use Doctrine\Common\Collections\ArrayCollection;

/**
 * Sdz\BlogBundle\Entity\Article
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 */
class Article
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

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

    /**
     * @var string $titre
     *
     * @ORM\Column(name="titre", type="string", length=255)
     * @Assert\MinLength(10)
     */
    private $titre;
}
```

```

    /**
 * @var text $contenu
 *
 * @ORM\Column(name="contenu", type="text")
 * @Assert\NotBlank()
 */
private $contenu;

    /**
 * @var string $auteur
 *
 * @ORM\Column(name="auteur", type="string", length=255)
 * @Assert\MinLength(2)
 */
private $auteur;

    /**
 * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image",
 * cascade={"persist"})
 * @Assert\Valid()
 */
private $image;

// ...
}

```

Vraiment pratique d'avoir les métadonnées Doctrine et les règles de validation au même endroit, n'est-ce pas ?



J'ai pris l'exemple ici d'une entité, car ce sera souvent le cas. Mais n'oubliez pas que vous pouvez mettre des règles de validation sur n'importe quel objet, qui n'est pas forcément une entité.

Syntaxe

Revenons un peu sur les annotations que l'on a ajoutées. Nous avons utilisé la forme simple, qui est construite comme ceci :

Code : Autre

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

Avec :

- La `Contrainte`, qui peut être, comme vous l'avez vu, `NotBlank` ou `MinLength`, etc. Nous voyons plus loin toutes les contraintes possibles.
- La `Valeur` entre parenthèses, qui est la valeur de l'option par défaut. En effet chaque contrainte a plusieurs options, dont une par défaut souvent intuitive. Par exemple, l'option par défaut de `MinLength` (« longueur minimale » en français) est évidemment la valeur de la longueur minimale que l'on veut appliquer, 10 pour le titre.

Mais on peut aussi utiliser la forme étendue qui permet de personnaliser la valeur de plusieurs options en même temps, comme ceci :

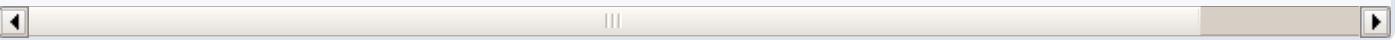
Code : Autre

```
@Assert\Contrainte(option1="valeur1", option2="valeur2", ...)
```

Les différentes options diffèrent d'une contrainte à une autre, mais voici un exemple avec la contrainte `MinLength` :

Code : Autre

```
@Assert\MinLength(limit=10, message="Le titre doit faire au moins {{ limit }} car")
```



 Oui, vous pouvez utiliser {{ limit }}, qui est en fait le nom de l'option que vous voulez faire apparaître dans le message.

Bien entendu, vous pouvez mettre plusieurs contraintes sur un même attribut. Par exemple pour un attribut numérique telle une note, on pourrait mettre les deux contraintes suivantes :

Code : PHP

```
<?php
/**
 * @Assert\Min(0)
 * @Assert\Max(20)
 */
private $note
```

Vous savez tout ! Il n'y a rien de plus à connaître sur les annotations. À part les contraintes existantes et leurs options, évidemment.

Liste des contraintes existantes

Voici un tableau qui regroupe la plupart des contraintes, à avoir sous la main lorsque vous définissez vos règles de validation ! Elles sont bien entendu toutes documentées, donc n'hésitez pas à vous référer à la [documentation officielle](#) pour toute information supplémentaire.

Toutes les contraintes disposent de l'option `message`, qui est le message à afficher lorsque la contrainte est violée. Je n'ai pas répété cette option dans les tableaux suivants, mais sachez qu'elle existe bien à chaque fois.

Contraintes de base :

Contrainte	Rôle	Options
NotBlank Blank	La contrainte NotBlank vérifie que la valeur soumise n'est ni une chaîne de caractères vide, ni NULL. La contrainte Blank fait l'inverse.	-
True False	La contrainte True vérifie que la valeur vaut true, 1 ou "1". La contrainte False vérifie que la valeur vaut false, 0 ou "0".	-
Type	La contrainte Type vérifie que la valeur est bien du type donné en argument.	type (option par défaut) : le type duquel doit être la valeur, parmi array, bool, int, object, etc.

Contraintes sur des chaînes de caractères :

Contrainte	Rôle	Options
Email	La contrainte Email vérifie que la valeur est une adresse e-mail valide.	checkMX (défaut : false) : si défini à true, Symfony2 va vérifier les MX de l'e-mail via la fonction checkdnsrr.

<code>MinLength</code>	La contrainte <code>MinLength</code> vérifie que la valeur donnée fait au moins X caractères de long.	<code>limit</code> (option par défaut) : le seuil de longueur, en nombre de caractères. <code>charset</code> (défaut : UTF-8) : le charset à utiliser pour calculer la longueur.
<code>Url</code>	La contrainte <code>Url</code> vérifie que la valeur est une adresse URL valide.	<code>protocols</code> (défaut : array('http', 'https')) : définit les protocoles considérés comme valides. Si vous voulez accepter les URL en <code>ftp://</code> , ajoutez-le à cette option.
<code>Regex</code>	La contrainte <code>Regex</code> vérifie la valeur par rapport à une regex.	<code>pattern</code> (option par défaut) : la regex à faire correspondre. <code>match</code> (défaut : <code>true</code>) : définit si la valeur doit (<code>true</code>) ou ne doit pas (<code>false</code>) correspondre à la regex.
<code>Ip</code>	La contrainte <code>Ip</code> vérifie que la valeur est une adresse IP valide.	<code>type</code> (défaut : 4) : version de l'IP à considérer. 4 pour IPv4, 6 pour IPv6, all pour toutes les versions, et <code>d'autres</code> .
<code>Language</code>	La contrainte <code>Language</code> vérifie que la valeur est un code de langage valide selon la norme.	-
<code>Locale</code>	La contrainte <code>Locale</code> vérifie que la valeur est une locale valide. Exemple : <code>fr</code> ou <code>fr_FR</code> .	-
<code>Country</code>	La contrainte <code>Country</code> vérifie que la valeur est un code pays en 2 lettres valide. Exemple : <code>fr</code> .	-

Contraintes sur les nombres :

Contrainte	Rôle	Options
<code>Max</code> <code>Min</code>	La contrainte <code>Max</code> vérifie que la valeur ne dépasse pas X. Min, que la valeur dépasse ce X.	<code>limit</code> (option par défaut) : la valeur seuil. <code>invalidMessage</code> : message d'erreur lorsque la valeur n'est pas un nombre.

Contraintes sur les dates :

Contrainte	Rôle	Options
<code>Date</code>	La contrainte <code>Date</code> vérifie que la valeur est un objet de type <code>Datetime</code> , ou une chaîne de caractères du type <code>YYYY-MM-DD</code> .	-
<code>Time</code>	La contrainte <code>Time</code> vérifie que la valeur est un objet de type <code>Datetime</code> , ou une chaîne de caractères du type <code>HH:MM:SS</code> .	-
<code>Datetime</code>	La contrainte <code>Datetime</code> vérifie que la valeur est un objet de type <code>Datetime</code> , ou une chaîne de caractères du type <code>YYYY-MM-DD HH:MM:SS</code> .	-

Contraintes sur les fichiers :

Contrainte	Rôle	Options
<code>File</code>	La contrainte <code>File</code> vérifie que la valeur est un fichier valide, c'est-à-dire soit une chaîne de caractères qui point vers un fichier existant, soit une instance de la classe <code>File</code> (ce qui inclut <code>UploadedFile</code>).	<code>maxSize</code> : la taille maximale du fichier. Exemple : 1M ou 1k. <code>mimeTypes</code> : <code>mimeType(s)</code> que le fichier doit avoir.
		<code>maxSize</code> : la taille maximale du fichier. Exemple : 1M ou 1k. <code>minWidth</code> / <code>minHeight</code> : la largeur et la hauteur minimales d'un image.

Image	<p>La contrainte <code>Image</code> vérifie que la valeur est valide selon la contrainte précédente <code>File</code> (dont elle hérite les options), sauf que les <code>mimeTypes</code> acceptés sont automatiquement définis comme ceux de fichiers images. Il est également possible de mettre des contraintes sur la hauteur <code>max</code> ou la largeur <code>max</code> de l'image.</p>	<code>maxWidth</code> : la largeur minimale et maximale que doit respecter l'image. <code>minHeight</code> / <code>maxHeight</code> : la hauteur minimale et maximale que doit respecter l'image.
--------------	---	--



Les noms de contraintes sont sensibles à la casse. Cela signifie que la contrainte `DateTime` existe, mais que `Datetime` ou `datetime` n'existent pas ! Soyez attentifs à ce détail pour éviter des erreurs inattendues. 😊

Déclencher la validation

Le service `Validator`

Comme je l'ai dit précédemment, ce n'est pas l'objet qui se valide tout seul, on doit déclencher la validation nous-mêmes. Ainsi, vous pouvez tout à fait assigner une valeur non valide à un attribut sans qu'aucune erreur ne se déclenche. Par exemple, vous pouvez faire `<?php $article->setTitre('abc')` alors que ce titre a moins de 10 caractères, il est invalide.

Pour valider l'objet, on passe par un acteur externe : le service `validator`. Ce service s'obtient comme n'importe quel autre service :

Code : PHP

```
<?php
// Depuis un contrôleur

$validator = $this->get('validator');
```

Ensuite, on doit demander à ce service de valider notre objet. Cela se fait grâce à la méthode `validate` du service. Cette méthode retourne un tableau qui est soit vide si l'objet est valide, soit rempli des différentes erreurs lorsque l'objet n'est pas valide. Pour bien comprendre, exécutez cette méthode dans un contrôleur :

Code : PHP

```
<?php
// Depuis un contrôleur

// ...

public function testAction()
{
    $article = new Article;

    $article->setDate(new \Datetime()); // Champ « date » OK
    $article->setTitre('abc'); // Champ « titre »
incorrect : moins de 10 caractères
    // $article->setContenu('blabla'); // Champ « contenu » incorrect
    : on ne le définit pas
    $article->setAuteur('A'); // Champ « auteur »
incorrect : moins de 2 caractères

    // On récupère le service validator
    $validator = $this->get('validator');

    // On déclenche la validation
    $listes_erreurs = $validator->validate($article);

    // Si le tableau n'est pas vide, on affiche les erreurs
    if(count($listes_erreurs) > 0) {
        return new Response(print_r($listes_erreurs, true));
    }
}
```

```
        } else {
            return new Response("L'article est valide !");
        }
    }
```

Vous pouvez vous amuser avec le contenu de l'entité Article pour voir comment réagit le validateur.

La validation automatique sur les formulaires

Vous devez savoir qu'en pratique on ne se servira que très peu du service validator nous-mêmes. En effet, le formulaire de Symfony2 le fait à notre place ! Nous venons de voir le fonctionnement du service validator pour comprendre comment l'ensemble marche, mais en réalité on l'utilisera très peu de cette manière.

Rappelez-vous le code pour la soumission d'un formulaire :

Code : PHP

```
<?php
// Depuis un contrôleur

if ($request->getMethod() == 'POST') {
    $form->bind($request);

    if ($form->isValid()) {
        // ...
    }
}
```

Avec la ligne 7, le formulaire \$form va lui-même faire appel au service validator, et valider l'objet qui vient d'être hydraté par le formulaire. Derrière cette ligne se cache donc le code que nous avons vu au paragraphe précédent. Les erreurs sont assignées au formulaire, et sont affichées dans la vue. Nous n'avons rien à faire, pratique !

Conclusion

Cette section a pour objectif de vous faire comprendre ce qu'il se passe déjà lorsque vous utilisez la méthode isValid d'un formulaire. De plus, vous savez qu'il est possible de valider un objet indépendamment de tout formulaire, en mettant la main à la pâte.

Encore plus de règles de validation

Valider depuis un getter

Vous le savez, un getter est une méthode qui commence le plus souvent par « get », mais qui peut également commencer par « is ». Le composant Validation accepte les contraintes sur les attributs, mais également sur les getters ! C'est très pratique, car vous pouvez alors mettre une contrainte sur une fonction, avec toute la liberté que cela vous apporte.

Tout de suite, un exemple d'utilisation :

Code : PHP

```
<?php
class Article
{
    // ...

    /**
     * @Assert\True()
     */
    public function isArticleValid()
    {
        // ...
    }
}
```

```
        return false;
    }
}
```

Cet exemple vraiment basique considère l'article comme non valide, car l'annotation `@Assert\True()` attend que la méthode retourne `true`, alors qu'elle retourne `false`. Vous pouvez l'essayer dans votre formulaire, vous verrez le message « Cette valeur doit être vraie » (message par défaut de l'annotation `True()`) qui s'affiche en haut du formulaire. C'est donc une erreur qui s'applique à l'ensemble du formulaire.

Mais il existe un moyen de déclencher une erreur liée à un champ en particulier, ainsi l'erreur s'affichera juste à côté de ce champ. Il suffit de nommer le getter « `is + le nom d'un attribut` » : par exemple `isTitre` si l'on veut valider le titre. Essayez par vous-mêmes le code suivant :

Code : PHP

```
<?php
class Article
{
    // ...

    /**
 * @Assert\True()
 */
    public function isTitre()
    {
        return false;
    }
}
```

Vous verrez que l'erreur « Cette valeur doit être vraie » s'affiche bien à côté du champ `titre`.

Bien entendu, vous pouvez faire plein de traitements et de vérifications dans cette méthode, ici j'ai juste mis `return false` pour l'exemple. Je vous laisse imaginer les possibilités.

Valider intelligemment un attribut objet

Derrière ce titre se cache une problématique toute simple : lorsque je valide un objet A, comment valider un objet B en attribut, d'après ses propres règles de validation ?

Il faut utiliser la contrainte `Valid`, qui va déclencher la validation du sous-objet B selon les règles de validation de cet objet B. Prenons un exemple :

Code : PHP

```
<?php
class A
{
    /**
 * @Assert\MinLength(5)
 */
    private $titre;

    /**
 * @Assert\Valid()
 */
    private $b;
}

class B
```

```
{
    /**
     * @Assert\Min(10)
     */
    private $nombre;
}
```

Avec cette règle, lorsqu'on déclenche la validation sur l'objet A, le service `validator` va valider l'attribut `titre` selon le `MinLength()`, puis va aller chercher les règles de l'objet B pour valider l'attribut `nombre` de B selon le `Min()`. N'oubliez pas cette contrainte, car valider un sous-objet n'est pas le comportement par défaut : sans cette règle dans notre exemple, vous auriez pu sans problème ajouter une instance de B qui ne respecte pas la contrainte de 10 minimum pour son attribut `nombre`. Vous pourriez donc rencontrer des petits soucis de logique si vous l'oubliez.

Valider depuis un Callback

L'objectif de la contrainte `Callback` est d'être personnalisable à souhait. En effet, vous pouvez parfois avoir besoin de valider des données selon votre propre logique, qui ne rentre pas dans un `Maxlength`.

L'exemple classique est la censure de mots non désirés dans un attribut texte. Reprenons notre `Article`, et considérons que l'attribut `contenu` ne peut pas contenir les mots « échec » et « abandon ». Voici comment mettre en place une règle qui va rendre invalide le contenu s'il contient l'un de ces mots :

Code : PHP

```
<?php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
use Doctrine\Common\Collections\ArrayCollection;
use Sdz\BlogBundle\Entity\Tag;

// On rajoute ce use pour le context :
use Symfony\Component\Validator\ExecutionContextInterface;

/**
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 *
 * N'oubliez pas cet Assert :
 * @Assert\Callback(methods={"contenuValide"})
 */
class Article
{
    // ...

    public function contenuValide(ExecutionContextInterface $context)
    {
        $mots_interdits = array('échec', 'abandon');

        // On vérifie que le contenu ne contient pas l'un des mots
        if (preg_match('#'.implode('|', $mots_interdits). '#', $this->getContenu())) {
            // La règle est violée, on définit l'erreur et son message
            // 1er argument : on dit quel attribut l'erreur concerne, ici « contenu »
            // 2e argument : le message d'erreur
            $context->addViolationAt('contenu', 'Contenu invalide car il contient un mot interdit.', array(), null);
        }
    }
}
```



L'annotation se définit ici sur la classe, et non sur une méthode ou un attribut, faites attention.

Vous auriez même pu aller plus loin en comparant des attributs entre eux, par exemple pour interdire le pseudo dans un mot de passe. L'avantage du Callback par rapport à une simple contrainte sur un getter, c'est de pouvoir ajouter plusieurs erreurs à la fois, en définissant sur quel attribut chacun se trouve grâce au premier argument de la méthode addViolationAt (en mettant contenu ou titre, etc). Souvent la contrainte sur un getter suffira, mais pensez à ce Callback pour les fois où vous serez limités. 😊

Valider un champ unique

Il existe une dernière contrainte très pratique : `UniqueEntity`. Cette contrainte permet de valider que la valeur d'un attribut est unique parmi toutes les entités existantes. Pratique pour vérifier qu'une adresse e-mail n'existe pas déjà dans la base de données par exemple.

Vous avez bien lu, j'ai parlé d'*entité*. En effet, c'est une contrainte un peu particulière, car elle ne se trouve pas dans le composant `Validator`, mais dans le Bridge entre Doctrine et Symfony2 (ce qui fait le lien entre ces deux bibliothèques). On n'utilisera donc pas `@Assert\UniqueEntity`, mais simplement `@UniqueEntity`. Il faut bien sûr en contrepartie faire attention de rajouter ce `use` à chaque fois que vous l'utilisez :

Code : Autre

```
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

Voici comment on pourrait, dans notre exemple avec `Article`, contraindre nos titres à être tous différents les uns des autres :

Code : PHP

```
<?php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
use Doctrine\Common\Collections\ArrayCollection;
use Sdz\BlogBundle\Entity\Tag;

// On rajoute ce use pour la contrainte :
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 */
* @UniqueEntity(fields="titre", message="Un article existe déjà avec
ce titre.")
*/
class Article
{
    // ... Les autres contraintes ne changent pas, pas même celle(s) sur
    l'attribut titre

    // Mais pour être logique, il faudrait aussi mettre la colonne
    titre en Unique pour Doctrine :
    /**
     * @ORM\Column(name="titre", type="string", length=255, unique=true)
     */
    private $titre;
}
```



Ici aussi, l'annotation se définit sur la classe, et non sur une méthode ou sur un attribut.

Valider selon nos propres contraintes

Vous commencez à vous habituer : avec Symfony2 il est possible de tout faire ! L'objectif de cette section est d'apprendre à créer notre propre contrainte, que l'on pourra utiliser en annotation : `@NotreContrainte`. L'avantage d'avoir sa propre contrainte est double :

- D'une part, c'est une contrainte réutilisable sur vos différents objets : on pourra l'utiliser sur `Article`, mais également sur `Commentaire`, etc. ;
- D'autre part, cela permet de placer le code de validation dans un objet externe... et surtout dans un service ! Indispensable, vous comprendrez.

Une contrainte est toujours liée à un validateur, qui va être en mesure de valider la contrainte. Nous allons donc les faire en deux étapes. Pour l'exemple, nous allons créer une contrainte `AntiFlood`, qui impose un délai de 15 secondes entre chaque message posté sur le site.

1. Créer la contrainte

Tout d'abord, il faut créer la contrainte en elle-même : c'est celle que nous appellerons en annotation depuis nos objets. Une classe de contrainte est vraiment très basique, toute la logique se trouvera en réalité dans le validateur. Je vous invite donc simplement à créer le fichier suivant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Validator/AntiFlood.php

namespace Sdz\BlogBundle\Validator;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class AntiFlood extends Constraint
{
    public $message = 'Vous avez déjà posté un message il y a moins de
15 secondes, merci d\'attendre un peu.';
}
```



L'annotation `@Annotation` est nécessaire pour que cette nouvelle contrainte soit disponible via les annotations dans les autres classes. En effet, toutes les classes ne sont pas des annotations, heureusement. 😊

Les options de l'annotation correspondent aux attributs publics de la classe d'annotation. Ici, on a l'attribut `message`, on pourra donc faire :

Code : Autre

```
@AntiFlood(message="Mon message personnalisé")
```

C'est tout pour la contrainte ! Passons au validateur.

2. Créer le validateur

C'est la contrainte qui décide par quel validateur elle doit se faire valider. Par défaut, une contrainte `Xxx` demande à se faire valider par le validateur `XxxValidator`. Créons donc le validateur `AntiFloodValidator` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Validator/AntiFloodValidator.php

namespace Sdz\BlogBundle\Validator;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class AntiFloodValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        // Pour l'instant, on considère comme flood tout message de
        // moins de 3 caractères
        if (strlen($value) < 3) {
            // C'est cette ligne qui déclenche l'erreur pour le
            // formulaire, avec en argument le message
            $this->context->addViolation($constraint->message);
        }
    }
}
```

C'est tout pour le validateur. Il n'est pas très compliqué non plus, il contient juste une méthode `validate()` qui permet de valider ou non la valeur. Son argument `$value` correspond à la valeur de l'attribut sur laquelle on a défini l'annotation. Par exemple, si l'on avait défini l'annotation comme ceci :

Code : Autre

```
/**
 * @AntiFlood()
 */
private $contenu;
```

... alors c'est tout logiquement le contenu de l'attribut `$contenu` au moment de la validation qui sera injecté en tant qu'argument `$value`.

La méthode `validate()` ne doit pas renvoyer `true` ou `false` pour confirmer que la valeur est valide ou non. Elle doit juste lever une `Violation` si la valeur est invalide. C'est ce qu'on a fait ici dans le cas où la chaîne fait moins de 3 caractères : on ajoute une violation, dont l'argument est le message d'erreur.

Sachez aussi que vous pouvez utiliser des messages d'erreur avec des paramètres. Par exemple :

"Votre message %string% est considéré comme flood". Pour définir ce paramètre `%string%` utilisé dans le message, il faut le passer dans le deuxième argument de la méthode `addViolation`, comme ceci :

Code : PHP

```
<?php
$this->context->addViolation($constraint->message, array('%string%' => $value));
```

Et voilà, vous savez créer votre propre contrainte ! Pour l'utiliser, c'est comme n'importe quelle autre annotation : on importe le

namespace de l'annotation, et on la met en commentaire juste avant l'attribut concerné. Voici un exemple sur l'entité Commentaire :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

namespace Sdz\BlogBundle\Entity\Commentaire;

use Sdz\BlogBundle\Validator\AntiFlood;

class Commentaire
{
    // ...

    /**
     * @ORM\Column(name="contenu", type="text")
     * @AntiFlood()
     */
    private $contenu;

    // ...
}
```

Votre annotation sera ainsi prise en compte au même titre qu'un `@Assert\MaxLength(x)` par exemple ! Mais si vous avez bien suivi, vous savez qu'on n'a pas encore vu le principal intérêt de nos propres contraintes : la validation par un service !

3. Transformer son validateur en service

Un service, c'est un objet qui remplit une fonction et auquel on peut accéder de presque n'importe où dans votre code Symfony2. Dans ce paragraphe, voyons comment s'en servir dans le cadre de nos contraintes de validation.



Quel est l'intérêt d'utiliser un service pour valider une contrainte ?

L'intérêt, comme on l'a vu dans les précédents chapitres sur les services, c'est qu'un service peut accéder à toutes sortes d'informations utiles. Il suffit de créer un service, de lui « injecter » les données, et il pourra ainsi s'en servir. Dans notre cas, on va lui injecter la requête et l'EntityManager comme données : il pourra ainsi valider notre contrainte non seulement à partir de la valeur `$value` d'entrée, mais également en fonction de paramètres extérieurs qu'on ira chercher dans la base de données !

3.1. Définition du service

Prenons un exemple pour bien comprendre le champ des possibilités. Il nous faut créer un service, en y injectant les services `request` et `entity_manager`, et en y apposant le tag `validator.constraint_validator`. Voici ce que cela donne, dans le fichier `services.yml` dans votre bundle :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdzblog.validator.antiflood:                                # Le
        nom du service
            class: Sdz\BlogBundle\Validator\AntiFloodValidator      # La
        classe du service, ici notre validateur déjà créé
            arguments: [@request, @doctrine.orm.entity_manager]   # Les
        données qu'on injecte au service : la requête et l'EntityManager
            scope: request                                         # Comme
```

```
on injecte la requête, on doit préciser ce scope
tags:
    - { name: validator.constraint_validator, alias:
sdzblog_antiflood } # C'est avec l'alias qu'on retrouvera le
service
```



Si le fichier `services.yml` n'existe pas déjà chez vous, c'est qu'il n'est pas chargé automatiquement. Pour cela, il faut faire une petite manipulation, je vous invite à lire le début du chapitre sur les services.

3.2. Modification de la contrainte

Maintenant que notre validateur est un service en plus d'être simplement un objet, nous devons adapter un petit peu notre code. Tout d'abord, modifions la contrainte pour qu'elle demande à se faire valider par le service d'alias `sdzblog_antiflood` et non plus simplement par l'objet classique `AntiFloodValidator`. Pour cela, il suffit de lui rajouter la méthode `validatedBy()` suivante (lignes 15 à 18) :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Validator/AntiFlood.php

namespace Sdz\BlogBundle\Validator;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class AntiFlood extends Constraint
{
    public $message = 'Vous avez déjà posté un message il y a moins de
15 secondes, merci d\'attendre un peu.';

    public function validatedBy()
    {
        return 'sdzblog_antiflood'; // Ici, on fait appel à l'alias du
service
    }
}
```

3.3. Modification du validateur

Enfin, il faut adapter notre validateur pour que d'une part il récupère les données qu'on lui injecte, grâce au constructeur, et d'autre part qu'il s'en serve tout simplement :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Validator/AntiFloodValidator.php

namespace Sdz\BlogBundle\Validator;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

use Doctrine\ORM\EntityManager;
use Symfony\Component\HttpFoundation\Request;
```

```

class AntiFloodValidator extends ConstraintValidator
{
    private $request;
    private $em;

    // Les arguments déclarés dans la définition du service arrivent
    // au constructeur
    // On doit les enregistrer dans l'objet pour pouvoir s'en
    resservir dans la méthode validate()
    public function __construct(Request $request, EntityManager $em)
    {
        $this->request = $request;
        $this->em = $em;
    }

    public function validate($value, Constraint $constraint)
    {
        // On récupère l'IP de celui qui poste
        $ip = $this->request->server->get('REMOTE_ADDR');

        // On vérifie si cette IP a déjà posté un message il y a moins
        de 15 secondes
        $isFlood = $this->em->getRepository('SdzBlogBundle:Commentaire')
            ->isFlood($ip, 15); // Bien entendu, il
        faudrait écrire cette méthode isFlood, c'est pour l'exemple

        if (strlen($value) < 3 && $isFlood) {
            // C'est cette ligne qui déclenche l'erreur pour le
            formulaire, avec en argument le message
            $this->context->addViolation($constraint->message);
        }
    }
}

```

Et voilà, nous venons de faire une contrainte qui s'utilise aussi facilement qu'une annotation, et qui pourtant fait un gros travail en allant chercher dans la base de données si l'IP courante envoie trop de messages. Un peu de travail à la création de la contrainte, mais son utilisation est un jeu d'enfant à présent !



Vous trouverez un brin plus d'informations sur la page de la documentation sur la création de contrainte, notamment comment faire une contrainte qui s'applique non pas à un attribut, mais à une classe entière.

Pour conclure

Vous savez maintenant valider dignement vos données, félicitations !

Le formulaire était le dernier point que vous aviez vraiment besoin d'apprendre. À partir de maintenant, vous pouvez créer un site internet en entier avec Symfony2, il ne manque plus que la sécurité à aborder, car pour l'instant, sur notre blog, tout le monde peut tout faire. Rendez-vous au prochain chapitre pour régler ce détail. 😊

En résumé

- Le composant `validator` permet de valider les données d'un objet suivant des règles définies.
- Cette validation est systématique lors de la soumission d'un formulaire : il est en effet impensable de laisser l'utilisateur entrer ce qu'il veut sans vérifier !
- Les règles de validation se définissent via les annotations directement à côté des attributs de la classe à valider. Vous pouvez bien sûr utiliser d'autres formats tels que le YAML ou le XML.
- Il est également possible de valider à l'aide de getters, de callbacks ou même de services. Cela rend la procédure de validation très flexible et très puissante.

Sécurité et gestion des utilisateurs

Dans ce chapitre, nous allons apprendre la sécurité avec Symfony2. C'est un chapitre assez technique, mais indispensable : à la fin nous aurons un espace membres fonctionnel et sécurisé !

Nous allons avancer en deux étapes : la première sera consacrée à la théorie de la sécurité sous Symfony2. Nécessaire, elle nous permettra d'aborder la deuxième étape : l'installation du bundle FOSUserBundle, qui viendra compléter notre espace membres.

Bonne lecture !

Authentification et autorisation

La sécurité sous Symfony2 est très poussée, vous pouvez la contrôler très finement, mais surtout très facilement. Pour atteindre ce but, Symfony2 a bien séparé deux mécanismes différents : l'authentification et l'autorisation. Prenez le temps de bien comprendre ces deux notions pour bien attaquer la suite du cours. 😊

Les notions d'authentification et d'autorisation

L'authentification

L'authentification est le processus qui va définir qui vous êtes, en tant que visiteur. L'enjeu est vraiment très simple : soit vous ne vous êtes pas identifié sur le site et vous êtes un anonyme, soit vous vous êtes identifié (via le formulaire d'identification ou via un cookie « Se souvenir de moi ») et vous êtes un membre du site. C'est ce que la procédure d'authentification va déterminer. Ce qui gère l'authentification dans Symfony2 s'appelle un *firewall*.

Ainsi vous pourrez sécuriser des parties de votre site internet juste en forçant le visiteur à être un membre authentifié. Si le visiteur l'est, le firewall va le laisser passer, sinon il le redirigera sur la page d'identification. Cela se fera donc dans les paramètres du firewall, nous les verrons plus en détail par la suite.

L'autorisation

L'autorisation est le processus qui va déterminer si vous avez le droit d'accéder à la ressource (la page) demandée. Il agit donc après le firewall. Ce qui gère l'autorisation dans Symfony2 s'appelle l'*access control*.

Par exemple, un membre identifié *lambda* aura accès à la liste de sujets d'un forum, mais ne peut pas supprimer de sujet. Seuls les membres disposant des droits d'administrateur peuvent, ce que l'*access control* va vérifier.

Exemples

Pour bien comprendre la différence entre l'authentification et l'autorisation, je reprends ici les exemples de [la documentation officielle](#), qui sont, je trouve, très intéressants et illustratifs. Dans ces exemples, vous distinguerez bien les différents acteurs de la sécurité.

Je suis anonyme, et je veux accéder à la page /foo qui ne requiert pas de droits

Dans cet exemple, un visiteur anonyme souhaite accéder à la page /foo. Cette page ne requiert pas de droits particuliers, donc tous ceux qui ont réussi à passer le firewall peuvent y avoir accès. La figure suivante montre le processus.

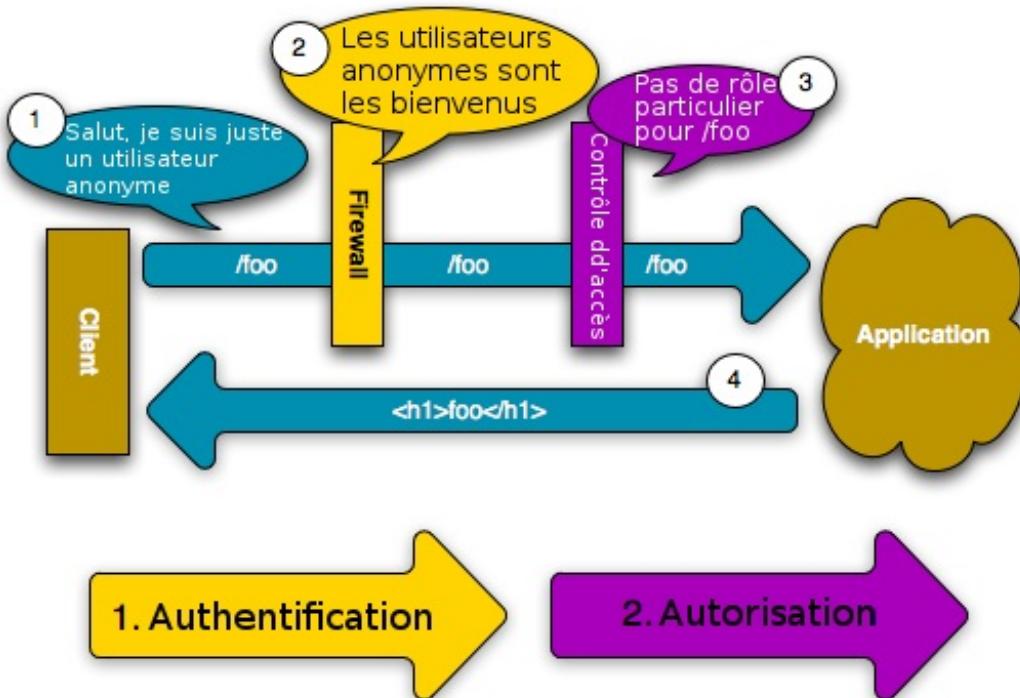


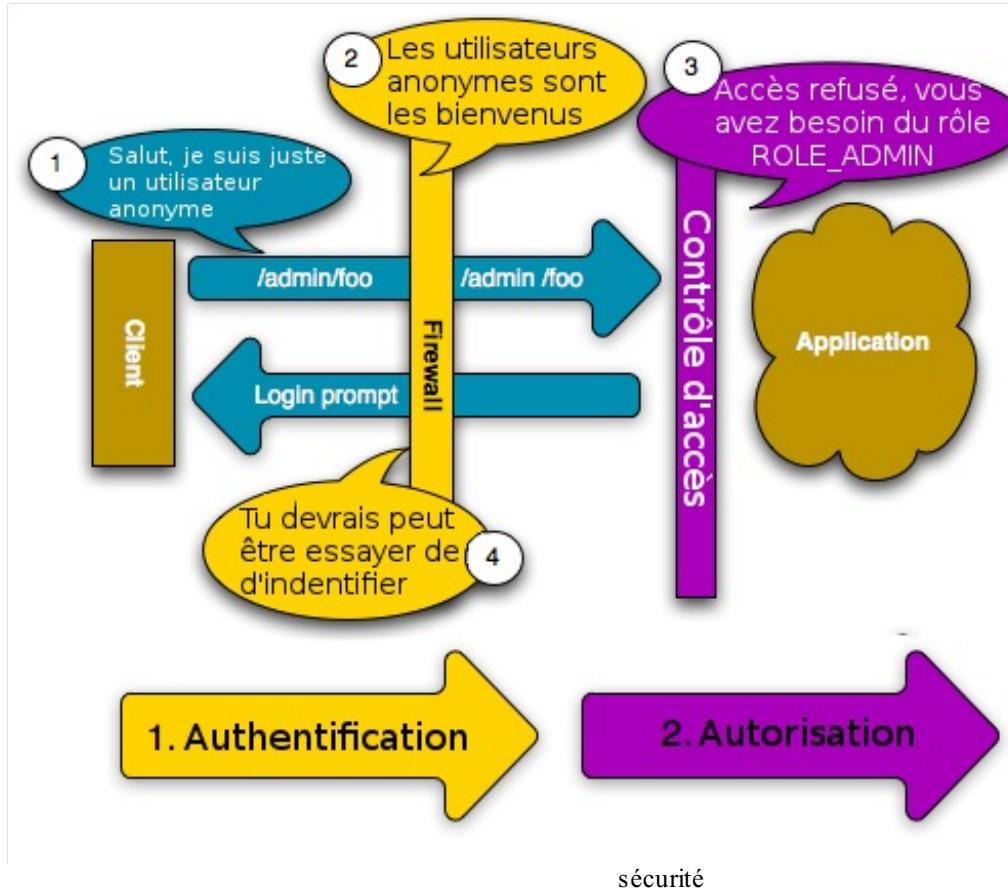
Schéma du processus de sécurité

Sur ce schéma, vous distinguez bien le firewall d'un côté et l'access control (contrôle d'accès) de l'autre. C'est très clair, mais reprenons-le ensemble pour bien comprendre :

1. Le visiteur n'est pas identifié, il est anonyme, et tente d'accéder à la page `/foo`.
2. Le firewall est configuré de telle manière qu'il n'est pas nécessaire d'être identifié pour accéder à la page `/foo`. Il laisse donc passer notre visiteur anonyme.
3. Le contrôle d'accès regarde si la page `/foo` requiert des droits d'accès : il n'y en a pas. Il laisse donc passer notre visiteur, qui n'a aucun droit particulier.
4. Le visiteur a donc accès à la page `/foo`.

Je suis anonyme, et je veux accéder à la page `/admin/foo` qui requiert certains droits

Dans cet exemple, c'est le même visiteur anonyme qui veut accéder à la page `/admin/foo`. Mais cette fois, la page `/admin/foo` requiert le rôle `ROLE_ADMIN` ; c'est un droit particulier, nous le verrons plus loin. Notre visiteur va se faire refuser l'accès à la page, la figure suivante montre comment.



Voici le processus pas à pas :

1. Le visiteur n'est pas identifié, il est toujours anonyme, et tente d'accéder à la page `/admin/foo`.
2. Le firewall est configuré de manière qu'il ne soit pas nécessaire d'être identifié pour accéder à la page `/admin/foo`. Il laisse donc passer notre visiteur.
3. Le contrôle d'accès regarde si la page `/admin/foo` requiert des droits d'accès : oui, il faut le rôle `ROLE_ADMIN`. Le visiteur n'a pas ce rôle, donc le contrôle d'accès lui interdit l'accès à la page `/admin/foo`.
4. Le visiteur n'a donc pas accès à la page `/admin/foo`, et se fait rediriger sur la page d'identification.

Je suis identifié, et je veux accéder à la page `/admin/foo` qui requiert certains droits

Cet exemple est le même que précédemment, sauf que cette fois notre visiteur est identifié, il s'appelle Ryan. Il n'est donc plus anonyme.

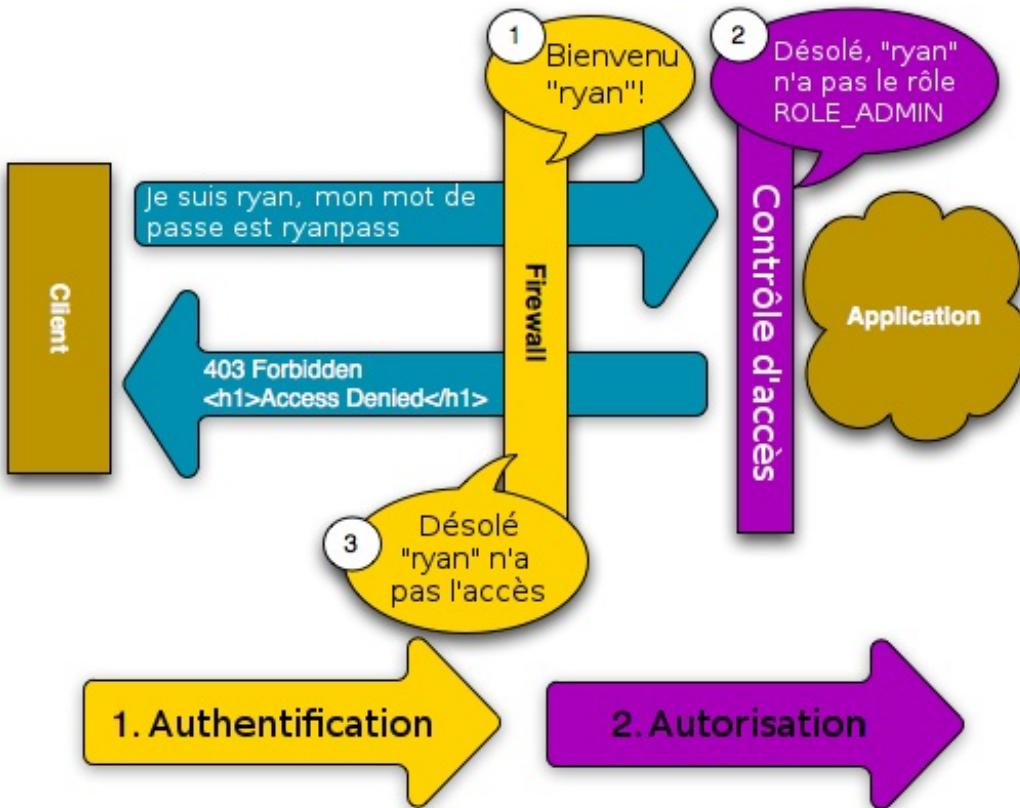


Schéma du processus de sécurité

sécurité

1. Ryan s'identifie et il tente d'accéder à la page /admin/foo. D'abord, le firewall confirme l'authentification de Ryan (c'est son rôle !). Visiblement c'est bon, il laisse donc passer Ryan.
2. Le contrôle d'accès regarde si la page /admin/foo requiert des droits d'accès : oui, il faut le rôle ROLE_ADMIN, que Ryan n'a pas. Il interdit donc l'accès à la page /admin/foo à Ryan.
3. Ryan n'a pas accès à la page /admin/foo non pas parce qu'il ne s'est pas identifié, mais parce que son compte utilisateur n'a pas les droits suffisants. Le contrôle d'accès lui affiche donc une page d'erreur lui disant qu'il n'a pas les droits suffisants.

Je suis identifié, et je veux accéder à la page /admin/foo qui requiert des droits que j'ai

Ici, nous sommes maintenant identifiés en tant qu'administrateur, on a donc le rôle ROLE_ADMIN ! Du coup, nous pouvons accéder à la page /admin/foo, comme le montre la figure suivante.

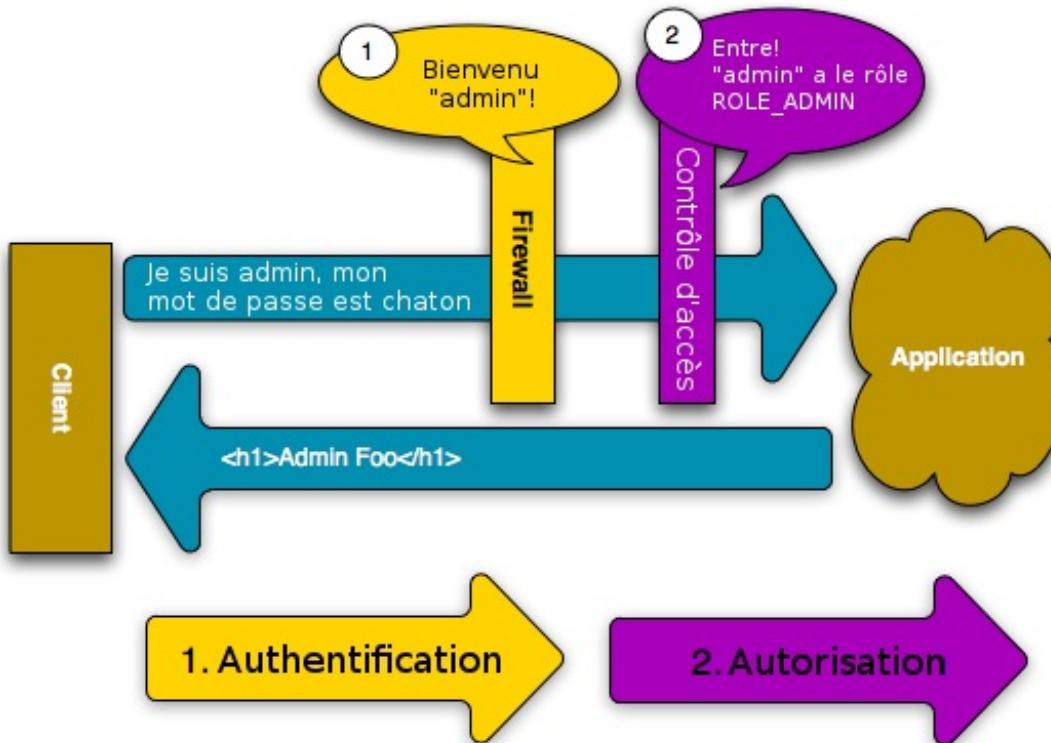


Schéma du processus

de sécurité

1. L'utilisateur admin s'identifie, et il tente d'accéder à la page /admin/foo. D'abord, le firewall confirme l'authentification d'admin. Ici aussi, c'est bon, il laisse donc passer admin.
2. Le contrôle d'accès regarde si la page /admin/foo requiert des droits d'accès : oui, il faut le rôle ROLE_ADMIN, qu'admin a bien. Il laisse donc passer l'utilisateur.
3. L'utilisateur admin a alors accès à la page /admin/foo, car il est identifié et il dispose des droits nécessaires.

Processus général

Lorsqu'un utilisateur tente d'accéder à une ressource protégée, le processus est finalement toujours le même, le voici :

1. Un utilisateur veut accéder à une ressource protégée ;
2. Le firewall redirige l'utilisateur au formulaire de connexion ;
3. L'utilisateur soumet ses informations d'identification (par exemple login et mot de passe) ;
4. Le firewall authentifie l'utilisateur ;
5. L'utilisateur authentifié renvoie la requête initiale ;
6. Le contrôle d'accès vérifie les droits de l'utilisateur, et autorise ou non l'accès à la ressource protégée.

Ces étapes sont simples, mais très flexibles. En effet, derrière le mot « authentification » se cache en pratique bien des méthodes : un formulaire de connexion classique, mais également l'authentification via Facebook, Google, etc., ou via les certificats X.509, etc. Bref, le processus reste toujours le même, mais les méthodes pour authentifier vos internautes sont nombreuses, et répondent à tous vos besoins. Et, surtout, elles n'ont pas d'impact sur le reste de votre code : qu'un utilisateur soit authentifié via Facebook ou un formulaire classique ne change rien à vos contrôleurs !

Première approche de la sécurité

Si les processus que nous venons de voir sont relativement simples, leur mise en place et leur configuration nécessitent un peu de travail.

Nous allons construire pas à pas la sécurité de notre application. Cette section commence donc par une approche théorique de la configuration de la sécurité avec Symfony2 (notamment l'authentification), puis on mettra en place un formulaire de connexion simple. On pourra ainsi s'identifier sur notre propre site, ce qui est plutôt intéressant ! Par contre, les utilisateurs ne seront pas encore liés à la base de données, on le verra un peu plus loin, avançons doucement.

Le fichier de configuration de la sécurité

La sécurité étant un point important, elle a l'honneur d'avoir son propre fichier de configuration. Il s'agit du fichier `security.yml`, situé dans le répertoire `app/config` de votre application. Je vous propose déjà d'y faire un petit nettoyage : supprimez les deux sections `login` et `secured_area` sous la section `firewalls`, elles concernent le bundle de démonstration que nous avons supprimé au début du cours. Votre fichier doit maintenant ressembler à ceci :

Code : YAML

```
# app/config/security.yml

jms_security_extra:
    secure_all_services: false
    expressions: true

security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    role_hierarchy:
        ROLE_ADMIN:           ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN,
        ROLE_ALLOWED_TO_SWITCH]

    providers:
        in_memory:
            memory:
                users:
                    user: { password: userpass, roles: [
        'ROLE_USER' ] }
                    admin: { password: adminpass, roles: [
        'ROLE_ADMIN' ] }

    firewalls:
        dev:
            pattern:  ^/(profiler|wdt)|css|images|js/
            security: false

    access_control:
        #- { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY,
        requires_channel: https }
```

Bien évidemment, rien de tout cela ne vous parle pour le moment. Rassurez-vous : à la fin du chapitre ce fichier ne vous fera plus peur. 😊 Pour le moment, décrivons rapidement chaque section de la configuration.

Section `jms_security_extra`

Code : YAML

```
jms_security_extra:
    secure_all_services: false
    expressions: true
```

Cette section concerne le bundle `JMSSecurityExtraBundle`, qui est livré par défaut avec Symfony2. Il apporte quelques petits plus à la sécurité, que nous verrons plus loin dans ce chapitre. Nous ne toucherons de toute façon pas à sa configuration, laissons cette section de côté pour l'instant.

Section encoders

Code : YAML

```
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
```

Un encodeur est un objet qui encode les mots de passe de vos utilisateurs. Cette section de configuration permet donc de modifier l'encodeur utilisé pour vos utilisateurs, et donc la façon dont sont encodés les mots de passe dans votre application.

Vous l'avez deviné, ici l'encodeur utilisé `plaintext` n'encode en réalité rien du tout. Il laisse en fait les mots de passe en clair, c'est pourquoi les mots de passe que nous verrons dans une section juste en dessous sont en clair. Évidemment, nous définirons par la suite un vrai encodeur, du type `sha512`, une méthode sûre !

Section role_hierarchy

Code : YAML

```
security:
    role_hierarchy:
        ROLE_ADMIN:             ROLE_USER
        ROLE_SUPER_ADMIN:       [ROLE_USER, ROLE_ADMIN,
        ROLE_ALLOWED_TO_SWITCH]
```

La notion de « rôle » est au centre du processus d'autorisation. On assigne un ou plusieurs rôles à chaque utilisateur, et chaque ressource nécessite un ou plusieurs rôles. Ainsi, lorsqu'un utilisateur tente d'accéder à une ressource, le contrôleur d'accès vérifie s'il dispose du ou des rôles requis par la ressource. Si c'est le cas, l'accès est accordé. Sinon, l'accès est refusé.

Cette section de la configuration dresse la hiérarchie des rôles. Ainsi, le rôle `ROLE_USER` est compris dans le rôle `ROLE_ADMIN`. Cela signifie que si votre page requiert le rôle `ROLE_USER`, et qu'un utilisateur disposant du rôle `ROLE_ADMIN` tente d'y accéder, il sera autorisé, car en disposant du rôle d'administrateur, il dispose également du rôle `ROLE_USER`.

Les noms des rôles n'ont pas d'importance, si ce n'est qu'ils doivent commencer par « `ROLE_` ».

Section providers

Code : YAML

```
security:
    providers:
        in_memory:
            memory:
                users:
                    user: { password: userpass, roles: [
                    'ROLE_USER' ] }
                    admin: { password: adminpass, roles: [
                    'ROLE_ADMIN' ] }
```

Un *provider* est un fournisseur d'utilisateurs. Les firewalls s'adressent aux providers pour récupérer les utilisateurs pour les identifier.

Pour l'instant vous pouvez le voir dans le fichier, un seul fournisseur est défini, nommé `in_memory` (encore une fois, le nom est arbitraire). C'est un fournisseur assez particulier dans le sens où les utilisateurs sont directement listés dans ce fichier de configuration, il s'agit des utilisateurs « `user` » et « `admin` ». Vous l'aurez compris, c'est un fournisseur de développement, pour

tester la couche sécurité sans avoir besoin d'une quelconque base de données derrière.

Je vous rassure, il existe d'autres types de fournisseurs que celui-ci. On utilisera notamment par la suite un fournisseur permettant de récupérer les utilisateurs dans la base de données, il est déjà bien plus intéressant.

Section firewalls

Code : YAML

```
security:  
    firewalls:  
        dev:  
            pattern:  ^/(_(profiler|wdt)|css|images|js)/  
            security: false
```

Comme on l'a vu précédemment, un firewall (ou pare-feu) cherche à vérifier que vous êtes bien celui que vous prétendez être. Ici, seul le pare-feu `dev` est défini, nous avons supprimé les autres pare-feu de démonstration. Ce pare-feu permet de désactiver la sécurité sur certaines URL, on en reparle plus loin.

Section access_control

Code : YAML

```
security:  
    access_control:  
        #- { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY,  
        requires_channel: https }
```

Comme on l'a vu, le contrôle d'accès (ou *access control* en anglais) va s'occuper de déterminer si le visiteur a les bons droits (rôles) pour accéder à la ressource demandée. Il y a différents moyens d'utiliser les contrôles d'accès :

- Soit ici depuis la configuration, en appliquant des règles sur des URL. On sécurise ainsi un ensemble d'URL en une seule ligne, par exemple toutes celles qui commencent par `/admin`.
- Soit directement dans les contrôleurs, en appliquant des règles sur les méthodes des contrôleurs. On peut ainsi appliquer des règles différentes selon des paramètres, vous êtes très libres.

Ces deux moyens d'utiliser la même protection par rôle sont très complémentaires, et offrent une flexibilité intéressante, on en reparle.

Mettre en place un pare-feu

Maintenant que nous avons survolé le fichier de configuration, vous avez une vue d'ensemble rapide de ce qu'il est possible de configurer. Parfait !

Il est temps de passer aux choses sérieuses, en mettant en place une authentification pour notre application. Nous allons le faire en deux étapes. La première est la construction d'un pare-feu, la deuxième est la construction d'un formulaire de connexion. Commençons.

1. Créer le pare-feu

Commençons par créer un pare-feu simple, que nous appellerons `main`, comme ceci :

Code : YAML

```
# app/Config/security.yml
```

```

security:
    firewalls:
        dev:
            pattern:  ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            pattern:    ^
            anonymous: true

```

Dans les trois petites lignes que nous venons de rajouter :

- main est le nom du pare-feu. Il s'agit juste d'un identifiant unique, mettez en réalité ce que vous voulez.
- pattern: ^/ est un masque d'URL. Cela signifie que toutes les URL commençant par « / » (c'est-à-dire notre site tout entier) sont protégées par ce pare-feu. On dit qu'elles sont derrière le pare-feu main.
- anonymous: true accepte les utilisateurs anonymes. Nous protégerons nos ressources grâce aux rôles.

 Le pare-feu main recoupe les URL du pare-feu dev, c'est vrai. En fait, un seul pare-feu peut agir sur une URL, et la règle d'attribution est la même que pour les routes : premier arrivé, premier servi ! En l'occurrence, le pare-feu dev est défini avant notre pare-feu main, donc une URL /css/... sera protégée par le pare-feu dev (car elle correspond à son pattern). Ce pare-feu désactive totalement la sécurité, au final les URL /css/... ne sont pas protégées du tout. 😊

Si vous actualisez n'importe quelle page de votre site, vous pouvez maintenant voir dans la barre d'outils en bas que vous êtes authentifiés en tant qu'anonyme, comme sur la figure suivante.



 Authentifié en tant qu'anonyme ? C'est pas un peu bizarre ça ?

Hé, hé ! en effet. En fait les utilisateurs anonymes sont *techniquement* authentifiés. Mais ils restent des anonymes, et si nous mettions la valeur du paramètre anonymous à false, on serait bien refusés. Pour distinguer les anonymes authentifiés des vrais membres authentifiés, il faudra jouer sur les rôles, on en reparle plus loin, ne vous inquiétez pas.

Bon, votre pare-feu est maintenant créé, mais bien sûr il n'est pas complet, il manque un élément indispensable pour le faire fonctionner : la méthode d'authentification. En effet, votre pare-feu veut bien protéger vos URL, mais il faut lui dire comment vérifier que vos visiteurs sont bien identifiés ! Et notamment, où trouver vos utilisateurs !

Définir une méthode d'authentification pour le pare-feu

Nous allons faire simple pour la méthode d'authentification : un bon vieux formulaire HTML. Pour configurer cela, c'est l'option form_login qu'il faut rajouter à notre pare-feu :

Code : YAML

```
# app/config/security.yml
```

```

security:
    firewalls:
        # ...
        main:
            pattern:    ^
            anonymous:  true
            provider:   in_memory
            form_login:
                login_path:  login
                check_path:  login_check
            logout:
                path:       logout
                target:     /blog

```

Expliquons les quelques nouvelles lignes :

- `provider: in_memory` est le fournisseur d'utilisateurs pour ce pare-feu. Comme je vous l'ai mentionné précédemment, un pare-feu a besoin de savoir où trouver ses utilisateurs, cela se fait par le biais de ce paramètre. La valeur `in_memory` correspond au fournisseur défini dans la section `providers` qu'on a vue précédemment.
- `form_login` est la méthode d'authentification utilisée pour ce pare-feu. Elle correspond à la méthode classique, via un formulaire HTML. Ses options sont les suivantes :
 - `login_path: login` correspond à la route du formulaire de connexion. En effet, ce formulaire est bien disponible à une certaine adresse, il s'agit ici de la route `login`, que nous définissons juste après.
 - `check_path: login_check` correspond à la route de validation du formulaire de connexion, c'est sur cette route que seront vérifiés les identifiants renseignés par l'utilisateur sur le formulaire précédent.
- `logout` rend possible la déconnexion. En effet, par défaut il est impossible de se déconnecter une fois authentifié. Ses options sont les suivantes :
 - `path` est le nom de la route à laquelle le visiteur doit aller pour être déconnecté. On va la définir plus loin.
 - `target` est l'URL vers laquelle sera redirigé le visiteur après sa déconnexion.

Je vous dois plus d'explications. Rappelez-vous, le processus est le suivant : lorsque le système de sécurité (ici, le pare-feu) initie le processus d'authentification, il va rediriger l'utilisateur sur le formulaire de connexion (la route `login`). On va créer ce formulaire juste après, il devra envoyer les valeurs vers la route (ici, `login_check`) qui va prendre en charge la soumission du formulaire.

Nous nous occupons du formulaire, mais c'est le système de sécurité de Symfony2 qui va s'occuper de la soumission de ce formulaire. Concrètement, nous allons définir un contrôleur à exécuter pour la route `login`, mais pas pour la route `login_check` ! Symfony2 va attraper la requête de notre visiteur sur la route `login_check`, et gérer lui-même l'authentification. En cas de succès, le visiteur sera authentifié. En cas d'échec, Symfony2 le renvoie vers notre formulaire de connexion.

Voici alors les trois routes à définir dans le fichier `routing.yml` :

Code : YAML

```

# app/config/routing.yml
# ...

login:
    pattern:    /login
    defaults:  { _controller: SdzUserBundle:Security:login }

login_check:
    pattern:    /login_check

logout:
    pattern:    /logout

```

Comme vous pouvez le voir, on ne définit pas de contrôleur pour les routes `login_check` et `logout`. Symfony2 va attraper tout seul les requêtes sur ces routes (grâce au gestionnaire d'événements, nous voyons cela dans [un prochain chapitre](#)).

Créer le bundle `SdzUserBundle`



Ce paragraphe n'est applicable que si vous ne disposez pas déjà d'un bundle `UserBundle`.

Cela ne vous a pas échappé, j'ai défini le contrôleur à exécuter sur la route `login` comme étant dans le bundle `SdzUserBundle`. En effet, la gestion des utilisateurs sur un site mérite amplement son propre bundle !

Je vous laisse générer ce bundle à l'aide de la commande suivante qu'on a déjà abordée :

Code : Console

```
php app/console generate:bundle
```

Si vous mettez `yes` pour importer automatiquement les routes du bundle généré, la commande va vous retourner ce message :

Code : Console

```
Importing the bundle routing resource: FAILED
```

```
The command was not able to configure everything automatically.  
You must do the following changes manually.
```

```
Bundle SdzUserBundle is already imported.
```

C'est parce qu'elle a détecté qu'on utilisait déjà une route vers ce bundle (celle qu'on vient de créer !), du coup elle n'a pas osé réimporter les routes du bundle. C'est de toute façon totalement inutile, pour l'instant on n'a pas de route dans ce bundle.

Avant de continuer, je vous propose un petit nettoyage, car le générateur a tendance à trop en faire. Vous pouvez donc supprimer allègrement :

- Le contrôleur `Controller/DefaultController.php`;
- Son répertoire de tests `Tests/Controller`;
- Son répertoire de vues `Resources/views/Default`;
- Le fichier de routes `Resources/config/routing.yml`.

Créer le formulaire de connexion

Il s'agit maintenant de créer le formulaire de connexion, disponible sur la route `login`, soit l'URL `/login`. Commençons par le contrôleur :

Code : PHP

```
<?php  
// src/Sdz/UserBundle/Controller/SecurityController.php;  
  
namespace Sdz\UserBundle\Controller;  
  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\Security\Core\SecurityContext;
```

```

class SecurityController extends Controller
{
    public function loginAction()
    {
        // Si le visiteur est déjà identifié, on le redirige vers
        l'accueil
        if ($this->get('security.context')-
>isGranted('IS_AUTHENTICATED_REMEMBERED')) {
            return $this->redirect($this->generateUrl('sdzblog_accueil'));
        }

        $request = $this->getRequest();
        $session = $request->getSession();

        // On vérifie s'il y a des erreurs d'une précédente soumission
        du formulaire
        if ($request->attributes-
>has(SecurityContext::AUTHENTICATION_ERROR)) {
            $error = $request->attributes-
>get(SecurityContext::AUTHENTICATION_ERROR);
        } else {
            $error = $session->get(SecurityContext::AUTHENTICATION_ERROR);
            $session->remove(SecurityContext::AUTHENTICATION_ERROR);
        }

        return $this->render('SdzUserBundle:Security:login.html.twig',
array(
    // Valeur du précédent nom d'utilisateur entré par
    l'internaute
    'last_username' => $session-
>get(SecurityContext::LAST_USERNAME),
    'error'           => $error,
));
    }
}

```

Ne vous laissez pas impressionner par le contrôleur, de toute façon vous n'avez pas à le modifier pour le moment. En réalité, il ne fait qu'afficher la vue du formulaire. Le code au milieu n'est là que pour récupérer les erreurs d'une éventuelle soumission précédente du formulaire. Rappelez-vous : c'est Symfony2 qui gère la soumission, et lorsqu'il y a une erreur dans l'identification, il redirige le visiteur vers ce contrôleur, en nous donnant heureusement l'erreur pour qu'on puisse lui afficher.

La vue pourrait être la suivante :

Code : HTML & Django

```

{# src/Sdz/UserBundle/Resources/views/Security/login.html.twig #-}

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

{%
    block body
%}

{# S'il y a une erreur, on l'affiche dans un joli cadre #}
{%
    if error %}
        <div class="alert alert-error">{{ error.message }}</div>
{%
    endif %}

{# Le formulaire, avec URL de soumission vers la route «
login_check » comme on l'a vu #}
<form action="{{ path('login_check') }}" method="post">
    <label for="username">Login :</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}"/>

    <label for="password">Mot de passe :</label>
    <input type="password" id="password" name="_password" />
    <br />

```

```
<input type="submit" value="Connexion" />  
</form>  
  
{ % endblock %}
```

La figure suivante montre le rendu du formulaire, accessible à l'adresse [/login](#).

Login :

Mot de passe :

Le formulaire de connexion

Lorsque j'entre de faux identifiants, l'erreur générée est celle visible à la figure suivante.

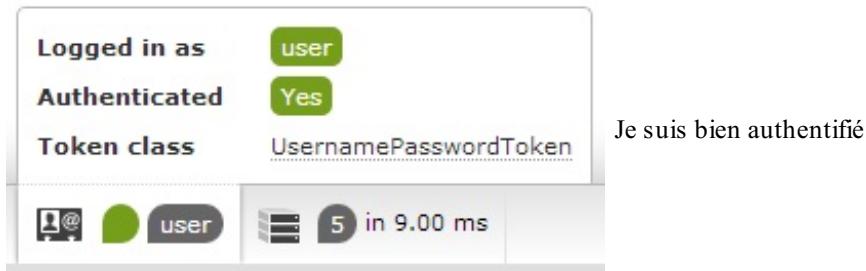
Bad credentials

Login :

 Mauvais identifiants

Mot de passe :

Enfin, lorsque j'entre les bons identifiants, la barre d'outils sur la page suivante m'indique bien que je suis authentifié en tant qu'utilisateur « user », comme le montre la figure suivante.



 Mais quels sont les bons identifiants ?

Il fallait lire attentivement le fichier de configuration qu'on a parcouru précédemment. Rappelez-vous, on a défini le fournisseur de notre pare-feu à `in_memory`, qui est défini quelques lignes plus haut dans le fichier de configuration. Ce fournisseur est particulier, dans le sens où il lit les utilisateurs directement dans cette configuration. On a donc deux utilisateurs possibles : « user » et « admin », avec pour mot de passe respectivement « userpass » et « adminpass ».

Voilà, notre formulaire de connexion est maintenant opérationnel. Vous trouverez plus d'informations pour le personnaliser [dans la documentation](#).

Les erreurs courantes

Il y a quelques pièges à connaître quand vous travaillerez plus avec la sécurité, en voici quelques-uns.

Ne pas oublier la définition des routes

Une erreur bête est d'oublier de créer les routes `login`, `login_check` et `logout`. Ce sont des routes obligatoires, et si vous les oubliez vous risquez de tomber sur des erreurs 404 au milieu de votre processus d'authentification.

Les pare-feu ne partagent pas

Si vous utilisez plusieurs pare-feu, sachez qu'ils ne partagent rien les uns avec les autres. Ainsi, si vous êtes authentifiés sur l'un, vous ne le serez pas forcément sur l'autre, et inversement. Cela permet d'accroître la sécurité lors d'un paramétrage complexe.

Bien mettre `/login_check` derrière le pare-feu

Vous devez vous assurer que l'URL du `check_path` (ici, `/login_check`) est bien derrière le pare-feu que vous utilisez pour le formulaire de connexion (ici, `main`). En effet, c'est la route qui permet l'authentification au pare-feu. Or, comme les pare-feu ne partagent rien, si cette route n'appartient pas au pare-feu que vous voulez, vous aurez droit à une belle erreur.

Dans notre cas, le pattern `^/` du pare-feu `main` prend bien l'URL `/login_check`, c'est donc OK.

Ne pas sécuriser le formulaire de connexion

En effet, si le formulaire est sécurisé, comment les nouveaux arrivants vont-ils pouvoir s'authentifier ? En l'occurrence, il faut faire attention que la page `/login` ne requière aucun rôle, on fera attention à cela lorsqu'on va définir les autorisations.



Cette erreur est vicieuse, car si vous sécurisez à tort l'URL `/login`, vous subirez une redirection infinie. En effet, Symfony2 considère que vous n'avez pas accès à `/login`, il vous redirige donc vers le formulaire pour vous authentifier, or il s'agit de la page `/login`, or vous n'avez pas accès à `/login`, etc.

De plus, si vous souhaitez interdire les anonymes sur le pare-feu `main`, le problème se pose également, car un nouvel arrivant sera anonyme et ne pourra pas accéder au formulaire de connexion. L'idée dans ce cas est de sortir le formulaire de connexion (la page `/login`) du pare-feu `main`. En effet, c'est le `check_path` qui doit obligatoirement appartenir au pare-feu, pas le formulaire en lui-même. Si vous souhaitez interdire les anonymes sur votre site (et uniquement dans ce cas), vous pouvez donc vous en sortir avec la configuration suivante :

Code : YAML

```
# app/config/security.yml
#
firewalls:
    # On crée un pare-feu uniquement pour le formulaire
    main_login:
        # Cette expression régulière permet de prendre /login (mais
        # pas /login_check !)
        pattern:    ^/login$
        anonymous: true # On autorise alors les anonymes sur ce
                        # pare-feu
    main:
        pattern:    ^
        anonymous: false
        # ...
```

En plaçant ce nouveau pare-feu *avant* notre pare-feu main, on sort le formulaire de connexion du pare-feu sécurisé. Nos nouveaux arrivants auront donc une chance de s'identifier !

Récupérer l'utilisateur courant

Pour récupérer les informations sur l'utilisateur courant, qu'il soit anonyme ou non, il faut utiliser le service `security.context`.

Ce service dispose d'une méthode `getToken()`, qui permet de récupérer la session de sécurité courante (à ne pas confondre avec la session classique, disponible elle via `$request->getSession()`). Ce token vaut `null` si vous êtes hors d'un pare-feu. Et si vous êtes derrière un pare-feu, alors vous pouvez récupérer l'utilisateur courant grâce à `$token->getUser()`.

Depuis le contrôleur ou un service

Voici concrètement comment l'utiliser :

Code : PHP

```
<?php

// On récupère le service
$security = $container->get('security.context');

// On récupère le token
$token = $security->getToken();

// Si la requête courante n'est pas derrière un pare-feu, $token
est null

// Sinon, on récupère l'utilisateur
$user = $token->getUser();

// Si l'utilisateur courant est anonyme, $user vaut « anon. »

// Sinon, c'est une instance de notre entité User, on peut
l'utiliser normalement
$user->getUsername();
```

Comme vous pouvez le voir, il y a pas mal de vérifications à faire, suivant les différents cas possibles. Heureusement, en pratique le contrôleur dispose d'un raccourci permettant d'automatiser cela, il s'agit de la méthode `$this->getUser()`. Cette méthode retourne :

- `null` si la requête n'est pas derrière un pare-feu, ou si l'utilisateur courant est anonyme ;
- Une instance de `User` le reste du temps (utilisateur authentifié derrière un pare-feu et non-anonyme).

Du coup, voici le code simplifié depuis un contrôleur :

Code : PHP

```
<?php
// Depuis un contrôleur

$user = $this->getUser();

if ($null === $user) {
    // Ici, l'utilisateur est anonyme ou l'URL n'est pas derrière un
pare-feu
} else {
    // Ici, $user est une instance de notre classe User
}
```

Depuis une vue Twig

Vous avez accès plus facilement à l'utilisateur directement depuis Twig. Vous savez que Twig dispose de quelques variables globales via la variable `{{ app }}` ; eh bien, l'utilisateur courant en fait partie, via `{{ app.user }}` :

Code : Django

```
Bonjour {{ app.user.username }} - {{ app.user.email }}
```

Au même titre que dans un contrôleur, attention à ne pas utiliser `{{ app.user }}` lorsque l'utilisateur n'est pas authentifié, car il vaut `null`.

Gestion des autorisations avec les rôles

La section précédente nous a amenés à réaliser une authentification opérationnelle. Vous avez un pare-feu, une méthode d'authentification par formulaire HTML, et deux utilisateurs. La couche **authentification** est complète !

Dans cette section, nous allons nous occuper de la deuxième couche de la sécurité : l'**autorisation**. C'est une phase bien plus simple à gérer heureusement, il suffit juste de demander tel(s) droit(s) à l'utilisateur courant (identifié ou non).

Définition des rôles

Rappelez-vous, on a croisé les rôles dans le fichier `security.yml`. La notion de rôle et autorisation est très simple : pour limiter l'accès à certaines pages, on va se baser sur les rôles de l'utilisateur. Ainsi, limiter l'accès au panel d'administration revient à limiter cet accès aux utilisateurs disposant du rôle `ROLE_ADMIN` (par exemple).

Tout d'abord, essayons d'imaginer les rôles dont on aura besoin dans notre application de blog. Je pense à :

- `ROLE_AUTEUR` : pour ceux qui ont le droit d'écrire des articles ;
- `ROLE_MODERATEUR` : pour ceux qui peuvent modérer les commentaires ;
- `ROLE_ADMIN` : pour ceux qui peuvent tout faire.

Maintenant l'idée est de créer une hiérarchie entre ces rôles. On va dire que les auteurs et les modérateurs sont bien différents, et que les admins ont les droits cumulés des auteurs et des modérateurs. Ainsi, pour limiter l'accès à certaines pages, on ne va pas faire « si l'utilisateur a `ROLE_AUTEUR` ou s'il a `ROLE_ADMIN`, alors il peut écrire un article ». Grâce à la définition de la hiérarchie, on peut faire simplement « si l'utilisateur a `ROLE_AUTEUR` ». Car un utilisateur qui dispose de `ROLE_ADMIN` dispose également de `ROLE_AUTEUR`, c'est une inclusion.

Ce sont ces relations, et uniquement ces relations, que nous allons inscrire dans le fichier `security.yml`. Voici donc comment décrire dans la configuration la hiérarchie qu'on vient de définir :

Code : YAML

```
# app/config/security.yml

security:
    role_hierarchy:
        ROLE_ADMIN:           [ROLE_AUTEUR, ROLE_MODERATEUR]          # Un
        admin hérite des droits d'auteur et de modérateur
        ROLE_SUPER_ADMIN:    [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH] # On
        garde ce rôle superadmin, il nous resservira par la suite
```

Remarquez que je n'ai pas utilisé le rôle `ROLE_USER`, qui n'est pas toujours utile. Avec cette hiérarchie, voici des exemples de tests que l'on peut faire :

- Si l'utilisateur a le rôle ROLE_AUTEUR, alors il peut écrire un article. Les auteurs et les admins peuvent donc le faire.
- Si l'utilisateur a le rôle ROLE_ADMIN, alors il peut supprimer un article. Seuls les admins peuvent donc le faire.

Tous ces tests nous permettront de limiter l'accès à nos différentes pages.



J'insiste sur le fait qu'on définit ici uniquement la hiérarchie entre les rôles, et non l'exhaustivité des rôles. Ainsi, on pourrait tout à fait avoir un rôle ROLE_TRUC dans notre application, mais que les administrateurs n'héritent pas.

Tester les rôles de l'utilisateur

Il est temps maintenant de tester concrètement si l'utilisateur courant dispose de tel ou tel rôle. Cela nous permettra de lui donner accès à la page, de lui afficher ou non un certain lien, etc. Laissez libre cours à votre imagination. 😊

Il existe quatre méthodes pour faire ce test : les annotations, le service security.context, Twig, et les contrôles d'accès. Ce sont quatre façons de faire exactement la même chose.

Utiliser directement le service security.context

Ce n'est pas le moyen le plus court, mais c'est celui par lequel passent les deux autres méthodes. Il faut donc que je vous en parle en premier !

Depuis votre contrôleur ou n'importe quel autre service, il vous faut accéder au service security.context et appeler la méthode isGranted, tout simplement. Par exemple dans notre contrôleur :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

// Pensez à rajouter ce use pour l'exception qu'on utilise
use Symfony\Component\HttpKernel\Exception\AccessDeniedHttpException;

// ...

class BlogController extends Controller
{
    public function ajouterAction($form)
    {
        // On teste que l'utilisateur dispose bien du rôle ROLE_AUTEUR
        if (!$this->get('security.context')->isGranted('ROLE_AUTEUR')) {
            // Sinon on déclenche une exception « Accès interdit »
            throw new AccessDeniedHttpException('Accès limité aux auteurs');
        }

        // ... Ici le code d'ajout d'un article qu'on a déjà fait
    }

    // ...
}
```

C'est tout ! Vous pouvez aller sur [/blog](#), mais impossible d'atteindre la page d'ajout d'un article sur [/blog/ajouter](#), car vous ne disposez pas (encore !) du rôle ROLE_AUTEUR, comme le montre la figure suivante.



Utiliser les annotations dans un contrôleur

Pour faire exactement ce qu'on vient de faire avec le service `security.context`, il existe un moyen bien plus rapide et joli : les annotations ! C'est ici qu'intervient le bundle `JMSSecurityExtraBundle`, présent et activé par défaut avec Symfony2. Pas besoin d'explication, c'est vraiment simple ; regardez le code :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

// Plus besoin de rajouter le use de l'exception dans ce cas
// Mais par contre il faut le use pour les annotations du bundle :
use JMS\SecurityExtraBundle\Annotation\Secure;

// ...

class BlogController extends Controller
{
    /**
     * @Secure(roles="ROLE_AUTEUR")
     */
    public function ajouterAction()
    {
        // Plus besoin du if avec le security.context, l'annotation
        // s'occupe de tout !
        // Dans cette méthode, vous êtes sûrs que l'utilisateur courant
        // dispose du rôle ROLE_AUTEUR

        // ... Ici le code d'ajout d'un article qu'on a déjà fait
    }

    // ...
}
```

Et voilà ! Grâce à l'annotation `@Secure`, on a sécurisé notre méthode en une seule ligne, vraiment pratique. Sachez que vous pouvez demander plusieurs rôles en même temps, en faisant `@Secure(roles="ROLE_AUTEUR, ROLE_MODERATEUR")`, qui demandera le rôle `ROLE_AUTEUR` et le rôle `ROLE_MODERATEUR` (ce n'est pas un « ou » !).

Pour vérifier simplement que l'utilisateur est authentifié, et donc qu'il n'est pas anonyme, vous pouvez utiliser le rôle spécial `IS_AUTHENTICATED_REMEMBERED`.

Sachez qu'il existe d'autres vérifications possibles avec l'annotation `@Secure`, je vous invite à jeter un œil à la documentation de `JMSSecurityExtraBundle`.

Depuis une vue Twig

Cette méthode est très pratique pour afficher du contenu différent selon les rôles de vos utilisateurs. Typiquement, le lien pour ajouter un article ne doit être visible que pour les membres qui disposent du rôle ROLE_AUTEUR (car c'est la contrainte que nous avons mise sur la méthode ajouterAction()).

Pour cela, Twig dispose d'une fonction `is_granted()` qui est en réalité un raccourci pour exécuter la méthode `isGranted()` du service `security.context`. La voici en application :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

{# ... #-}

{# On n'affiche le lien « Ajouter un article » qu'aux auteurs
(et admins, qui héritent du rôle auteur) #-}
{%- if is_granted('ROLE_AUTEUR') %}
  <a href="{{ path('sdzblog_ajouter') }}>Ajouter un article</a>
{%- endif %}

{# ... #}
```

Utiliser les contrôles d'accès

La méthode de l'annotation permet de sécuriser *une méthode de contrôleur*. La méthode avec Twig permet de sécuriser *l'affichage*. La méthode des contrôles d'accès permet de sécuriser des *URL*. Elle se configure dans le fichier de configuration de la sécurité, c'est la dernière section. Voici par exemple comment sécuriser tout un panel d'administration (des pages dont l'URL commence par /admin) en une seule ligne :

Code : YAML

```
# app/config/security.yml

security:
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
```

Ainsi, toutes les URL qui correspondent au path (ici, toutes celles qui commencent par /admin) requièrent le rôle ROLE_ADMIN.

C'est une méthode complémentaire des autres. Elle permet également de sécuriser vos URL par IP ou par canal (http ou https), grâce à des options :

Code : YAML

```
# app/config/security.yml

security:
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN, ip: 127.0.0.1,
            requires_channel: https }
```

Pour conclure sur les méthodes de sécurisation

Symfony2 offre plusieurs moyens de sécuriser vos ressources (méthode de contrôleur, affichage, URL). N'hésitez pas à vous servir de la méthode la plus appropriée pour chacun de vos besoins. C'est la complémentarité des méthodes qui fait l'efficacité de la sécurité avec Symfony2.



Pour tester les sécurités qu'on met en place, n'hésitez pas à charger vos pages avec les deux utilisateurs « user » et « admin ». L'utilisateur admin ayant le rôle ROLE_ADMIN, il a les droits pour ajouter un article et voir le lien d'ajout. Pour vous déconnecter d'un utilisateur, allez sur /logout.

Utiliser des utilisateurs de la base de données

Pour l'instant, nous n'avons fait qu'utiliser les deux pauvres utilisateurs définis dans le fichier de configuration. C'était pratique pour faire nos premiers tests, car ils ne nécessitent aucun paramétrage particulier. Mais maintenant, passons à la vitesse supérieure et enregistrons nos utilisateurs en base de données !

Qui sont les utilisateurs ?

Dans Symfony2, un utilisateur est un objet qui implémente l'interface `UserInterface`, c'est tout. N'hésitez pas à aller voir à quoi ressemble cette interface, il n'y a en fait que cinq méthodes obligatoires, ce n'est pas grand-chose.

Heureusement il existe également une classe `User` qui implémente cette interface. Les utilisateurs que nous avons actuellement sont des instances de cette classe.

Créons notre classe d'utilisateurs

En vue d'enregistrer nos utilisateurs en base de données, il nous faut créer notre propre classe utilisateur, qui sera également une entité pour être persistée. Je vous invite donc à générer directement une entité `User` au sein du bundle `SdzUserBundle`, grâce au générateur de Doctrine (`php app/console doctrine:generate:entity`), avec les attributs minimum suivants (tirés de l'interface) :

- `username` : c'est l'identifiant de l'utilisateur au sein de la couche sécurité. Cela ne nous empêchera pas d'utiliser également un id numérique pour notre entité, c'est plus simple pour nous ;
- `password` : le mot de passe ;
- `salt` : le sel, pour encoder le mot de passe, on en reparle plus loin ;
- `roles` : un tableau (attention à bien le définir comme tel lors de la génération) contenant les rôles de l'utilisateur.

Voici la classe que j'obtiens :

Code : PHP

```
<?php
// src/Sdz/UserBundle/Entity/User.php

namespace Sdz\UserBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
*
* @ORM\Entity(repositoryClass="Sdz\UserBundle\Entity\UserRepository")
*/
class User
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(name="username", type="string", length=255,
     * unique=true)
     */
    private $username;
```

```
 /**
 * @ORM\Column(name="password", type="string", length=255)
 */
 private $password;

 /**
 * @ORM\Column(name="salt", type="string", length=255)
 */
 private $salt;

 /**
 * @ORM\Column(name="roles", type="array")
 */
 private $roles;

 public function __construct()
 {
     $this->roles = array();
 }

 public function getId()
 {
     return $this->id;
 }

 public function setUsername($username)
 {
     $this->username = $username;
     return $this;
 }

 public function getUsername()
 {
     return $this->username;
 }

 public function setPassword($password)
 {
     $this->password = $password;
     return $this;
 }

 public function getPassword()
 {
     return $this->password;
 }

 public function setSalt($salt)
 {
     $this->salt = $salt;
     return $this;
 }

 public function getSalt()
 {
     return $this->salt;
 }

 public function setRoles(array $roles)
 {
     $this->roles = $roles;
     return $this;
 }

 public function getRoles()
 {
     return $this->roles;
 }
```

```
public function eraseCredentials()
{
}
```

 J'ai rajouté un constructeur pour définir une valeur par défaut (`array()`) à l'attribut `$roles`. J'ai également défini l'attribut `username` comme étant unique, car c'est l'identifiant qu'utilise la couche sécurité, il est donc obligatoire qu'il soit unique. Enfin, j'ai ajouté la méthode `eraseCredentials()`, vide pour l'instant mais obligatoire de par l'interface suivante.

Et pour que Symfony2 l'accepte comme classe utilisateur de la couche sécurité, il faut qu'on implémente l'interface `UserInterface` :

Code : PHP

```
<?php
// src/Sdz/UserBundle/Entity/User.php

use Symfony\Component\Security\Core\User\UserInterface;

class User implements UserInterface
{
    // ...
}
```

Et voilà, nous avons une classe prête à être utilisée !



Et bien sûr, exécutez un petit `php app/console doctrine:schema:update` pour mettre à jour la base de données avec cette nouvelle entité.

Créons quelques utilisateurs de test

Pour s'amuser avec notre nouvelle entité `User`, il faut créer quelques instances dans la base de données. Réutilisons ici les *fixtures*, voici ce que je vous propose :

Code : PHP

```
<?php
// src/Sdz/UserBundle/DataFixtures/ORM/Users.php

namespace Sdz\UserBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Sdz\UserBundle\Entity\User;

class Users implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        // Les noms d'utilisateurs à créer
        $noms = array('winzou', 'John', 'Talus');

        foreach ($noms as $i => $nom) {
            // On crée l'utilisateur
            $users[$i] = new User;
```

```

    // Le nom d'utilisateur et le mot de passe sont identiques
    $users[$i]->setUsername($nom);
    $users[$i]->setPassword($nom);

    // Le sel et les rôles sont vides pour l'instant
    $users[$i]->setSalt('');
    $users[$i]->setRoles(array());

    // On le persiste
    $manager->persist($users[$i]);
}

// On déclenche l'enregistrement
$manager->flush();
}
}

```

Exécutez cette fois la commande :

Code : Console

```
php app/console doctrine:fixtures:load
```

Et voilà, nous avons maintenant trois utilisateurs dans la base de données.

Définissons l'encodeur pour notre nouvelle classe d'utilisateurs

Ce n'est pas un piège mais presque, rappelez-vous l'encodeur défini pour nos précédents utilisateurs spécifiait la classe `User` utilisée. Or maintenant nous allons nous servir d'une autre classe, il s'agit de `Sdz\UserBundle\Entity\User`. Il est donc obligatoire de définir quel encodeur utiliser pour notre nouvelle classe. Comme nous avons mis les mots de passe en clair dans les *fixtures*, nous devons également utiliser l'encodeur `plaintext`, qui n'encode pas les mots de passe mais les laisse en clair, c'est plus simple pour nos tests.

Ajoutez donc cet encodeur dans la configuration, juste en dessous de celui existant :

Code : YAML

```

# app/config/security.yml

security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
        Sdz\UserBundle\Entity\User: plaintext

```

Définissons le fournisseur d'utilisateurs

On en a parlé plus haut, il faut définir un fournisseur (*provider*) pour que le pare-feu puisse identifier et récupérer les utilisateurs.

Qu'est-ce qu'un fournisseur d'utilisateurs, concrètement ?

Un fournisseur d'utilisateurs est une classe qui implémente l'interface `UserProviderInterface`, qui contient juste trois méthodes :

- `loadUserByUsername($username)`, qui charge un utilisateur à partir d'un nom d'utilisateur ;
- `refreshUser($user)`, qui rafraîchit un utilisateur avec les valeurs d'origine ;

- `supportsClass()`, qui détermine quelle classe d'utilisateurs gère le fournisseur.

Vous pouvez le constater, un fournisseur ne fait pas grand-chose, à part charger ou rafraîchir les utilisateurs.

Symfony2 dispose déjà de trois types de fournisseurs, qui implémentent tous l'interface précédente évidemment, les voici :

- `memory` utilise les utilisateurs définis dans la configuration, c'est celui qu'on a utilisé jusqu'à maintenant ;
- `entity` utilise de façon simple une entité pour fournir les utilisateurs, c'est celui qu'on va utiliser ;
- `id` permet d'utiliser un service quelconque en tant que fournisseur, en précisant le nom du service.

Créer notre fournisseur `entity`

Il est temps de créer le fournisseur `entity` pour notre entité. Celui-ci existe déjà dans Symfony2, nous n'avons donc pas de code à faire, juste un peu de configuration. On va l'appeler « `main` », un nom arbitraire. Voici comment le déclarer :

Code : YAML

```
# app/config/security.yml

security:
    providers:
        # ... vous pouvez supprimer le fournisseur « in_memory »
        # Et voici notre nouveau fournisseur :
        main:
            entity: { class: Sdz\UserBundle\Entity\User, property:
username }
```

Il y a deux paramètres à préciser pour le fournisseur :

- La classe à utiliser évidemment, il s'agit pour le fournisseur de savoir quel repository Doctrine récupérer pour ensuite charger nos entités ;
- L'attribut de la classe qui sert d'identifiant, on utilise `username`, donc on le lui dit.

Dire au pare-feu d'utiliser le nouveau fournisseur

Maintenant que notre fournisseur existe, il faut demander au pare-feu de l'utiliser lui, et non l'ancien fournisseur `in_memory`. Pour cela, modifions simplement la valeur du paramètre `provider`, comme ceci :

Code : YAML

```
# app/config/security.yml

security:
    firewalls:
        main:
            pattern:  ^
            anonymous: true
            provider: main # On change cette valeur
            # ... reste de la configuration du pare-feu
```



Vous trouverez encore plus d'informations sur ce type de fournisseur dans la documentation.

Manipuler vos utilisateurs

La couche sécurité est maintenant pleinement opérationnelle et utilise des utilisateurs stockés en base de données. C'est parfait !



Vous voulez faire un formulaire d'inscription ? Modifier vos utilisateurs ? Changer leurs rôles ?

Je pourrais vous expliquer comment le faire, mais en réalité vous savez déjà le faire !

L'entité `User` que nous avons créée est une entité tout à fait comme les autres. À ce stade du cours vous savez ajouter, modifier et supprimer des articles de blog, alors il en va de même pour cette nouvelle entité qui représente vos utilisateurs.

Bref, faites-vous confiance, vous avez toutes les clés en main pour manipuler entièrement vos utilisateurs.

Cependant, toutes les pages d'un espace membres sont assez classiques : inscription, mot de passe perdu, modification du profil, etc. Tout cela est du déjà-vu. Et si c'est déjà vu, il existe déjà certainement un bundle pour cela. Et je vous le confirme, il existe même un excellent bundle, il s'agit de `FOSUserBundle` et je vous propose de l'installer !

Utiliser `FOSUserBundle`

Comme vous avez pu le voir, la sécurité fait intervenir de nombreux acteurs et demande pas mal de travail de mise en place. C'est normal, c'est un point sensible d'un site internet. Heureusement, d'autres développeurs talentueux ont réussi à nous faciliter la tâche en créant un bundle qui gère une partie de la sécurité !

Ce bundle s'appelle `FOSUserBundle`, il est très utilisé par la communauté Symfony2 car vraiment bien fait, et surtout répondant à un besoin vraiment basique d'un site internet : l'authentification des membres.

Je vous propose donc d'installer ce bundle dans la suite de cette section. Cela n'est en rien obligatoire, vous pouvez tout à fait continuer avec le `User` qu'on vient de développer, cela fonctionne tout aussi bien !

Installation de `FOSUserBundle`

Télécharger le bundle

Le bundle `FOSUserBundle` est hébergé sur GitHub, comme beaucoup de bundles et projets Symfony2. Sa page est ici : <https://github.com/FriendsOfSymfony/FOSUserBundle>.

Mais pour ajouter ce bundle, vous l'avez compris, il faut utiliser Composer ! Commencez par déclarer cette nouvelle dépendance dans votre fichier `composer.json` :

Code : JavaScript

```
// composer.json
{
    // ...
    "require": {
        // ...
        "friendsofsymfony/user-bundle": "dev-master"
    }
    // ...
}
```

Ensuite, il faut dire à Composer d'installer cette nouvelle dépendance :

Code : Console

```
php composer.phar update friendsofsymfony/user-bundle
```



L'argument après la commande `update` permet de dire à Composer de ne mettre à jour que cette dépendance. Ici, cela permet de ne mettre à jour que `FOSUserBundle`, et pas les autres dépendances. C'est plus rapide, mais si vous voulez tout mettre à jour, supprimez simplement ce paramètre. 😊

Activer le bundle

Si vos souvenirs sont bons, vous devriez savoir qu'un bundle ne s'active pas tout seul, il faut aller l'enregistrer dans le noyau de Symfony2. Pour cela, ouvrez le fichier `app/AppKernel.php` pour enregistrer le bundle :

Code : PHP

```
<?php
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        ...
        new FOS\UserBundle\FOSUserBundle(),
    );
}
```

C'est bon, le bundle est bien enregistré. Mais inutile d'essayer d'accéder à votre application Symfony2 maintenant, elle ne marchera pas. Il faut en effet faire un peu de configuration et de personnalisation avant de pouvoir tout remettre en marche.

Hériter `FOSUserBundle` depuis notre `SdzUserBundle`

`FOSUserBundle` est un bundle générique évidemment, car il doit pouvoir s'adapter à tout type d'utilisateur de n'importe quel site internet. Vous imaginez bien que, du coup, ce n'est pas un bundle prêt à l'emploi directement après son installation ! Il faut donc s'atteler à le personnaliser afin de faire correspondre le bundle à nos besoins. Cette personnalisation passe par l'héritage de bundle.

C'est une fonctionnalité intéressante qui va nous permettre de personnaliser facilement et proprement le bundle que l'on vient d'installer. L'héritage de bundle est même très simple à réaliser. Prenez le fichier `SdzUserBundle.php` qui représente notre bundle, et modifiez-le comme suit :

Code : PHP

```
<?php
// src/Sdz/UserBundle/SdzUserBundle.php

namespace Sdz\UserBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class SdzUserBundle extends Bundle
{
    public function getParent()
    {
        return 'FOSUserBundle';
    }
}
```

Et c'est tout ! On a juste rajouté cette méthode `getParent()`, et Symfony2 va savoir gérer le reste. 😊

Modifier notre entité User

Bien que nous ayons déjà créé une entité `User`, ce nouveau bundle en contient une plus complète, qu'on va utiliser avec plaisir plutôt que de tout recoder nous-mêmes. Ici on va hériter de l'entité `User` du bundle, depuis notre entité `User` de notre bundle. En fait, notre entité ne contient plus grand-chose au final, voici ce que cela donne :

Code : PHP

```
<?php
// src/Sdz/UserBundle/Entity/User.php

namespace Sdz\UserBundle\Entity;

use FOS\UserBundle\Entity\User as BaseUser;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="sdz_user")
 */
class User extends BaseUser
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;
}
```



Plus besoin d'implémenter `UserInterface`, car on hérite de l'entité `User` du bundle `FOSUserBundle`, qui, elle, implémente cette interface. 😊

Alors c'est joli, mais pourquoi est-ce que l'on a fait cela ? En fait, le bundle `FOSUserBundle` ne définit pas vraiment l'entité `User`, il définit une *mapped superclass* ! Un nom un peu barbare, juste pour dire que c'est une entité abstraite, et qu'il faut en hériter pour faire une vraie entité. C'est donc ce que nous venons juste de faire.

Cela permet en fait de garder la main sur notre entité. On peut ainsi lui ajouter des attributs (selon vos besoins), en plus de ceux déjà définis. Pour information, les attributs qui existent déjà sont :

- `username` : nom d'utilisateur avec lequel l'utilisateur va s'identifier ;
- `email` : l'adresse e-mail ;
- `enabled` : true ou false suivant que l'inscription de l'utilisateur a été validée ou non (dans le cas d'une confirmation par e-mail par exemple) ;
- `password` : le mot de passe de l'utilisateur ;
- `lastLogin` : la date de la dernière connexion ;
- `locked` : si vous voulez désactiver des comptes ;
- `expired` : si vous voulez que les comptes expirent au-delà d'une certaine durée.

Je vous en passe certains qui sont plus à un usage interne. Sachez tout de même que vous pouvez tous les retrouver dans [la définition](#) de la *mapped superclass*. C'est un fichier de mapping XML, l'équivalent des annotations qu'on utilise de notre côté.

Vous pouvez rajouter dès maintenant des attributs à votre entité `User`, comme vous savez le faire depuis la partie Doctrine2.

Configurer le bundle

Ensuite, nous devons définir certains paramètres obligatoires au fonctionnement de `FOSUserBundle`. Ouvrez votre

config.yml et ajoutez la section suivante :

Code : YAML

```
# app/config/config.yml

# ...

fos_user:
    db_driver: orm # Le type de BDD à utiliser, nous utilisons l'ORM Doctrine depuis le début
    firewall_name: main # Le nom du firewall derrière lequel on utilisera ces utilisateurs
    user_class: Sdz\UserBundle\Entity\User # La classe de l'entité User que nous utilisons
```

Et voilà, on a bien installé FOSUserBundle ! Avant d'aller plus loin, créons la table User et ajoutons quelques membres pour les tests.

Mise à jour de la table User

Il faut mettre à jour la table des utilisateurs, vu les modifications que l'on vient de faire. D'abord, allez la vider depuis phpMyAdmin, puis exécutez la commande `php app/console doctrine:schema:update --force`. Et voilà, votre table est créée !

On a fini d'initialiser le bundle. Bon, bien sûr pour l'instant Symfony2 ne l'utilise pas encore, il manque un peu de configuration, attaquons-la.

Configuration de la sécurité pour utiliser le bundle

Maintenant on va reprendre notre configuration de la sécurité, pour utiliser tous les outils fournis par le bundle dès que l'on peut. Reprenez le security.yml sous la main, et c'est parti !

L'encodeur

Il est temps d'utiliser un vrai encodeur pour nos utilisateurs, car il est bien sûr hors de question de stocker leur mot de passe en clair ! On utilise couramment la méthode sha512. Modifiez donc l'encodeur de notre classe comme ceci :

Code : YAML

```
# app/config/security.yml

security:
    encoders:
        Sdz\UserBundle\Entity\User: sha512
```

Le fournisseur

Le bundle inclut son propre fournisseur, qui utilise notre entité User mais avec ses propres outils. Vous pouvez donc modifier notre fournisseur main comme suit :

Code : YAML

```
# app/config/security.yml
```

```

security:
    # ...

providers:
    main:
        id: fos_user.user_provider.username

```

Dans cette configuration, `fos_user.user_manager` est le nom du service fourni par le bundle FOSUB.

Le pare-feu

Notre pare-feu était déjà pleinement opérationnel. Étant donné que nous n'avons pas changé le nom du fournisseur associé, la configuration du pare-feu est déjà à jour. Nous n'avons donc rien à modifier ici.

On va juste en profiter pour activer la possibilité de « Se souvenir de moi » à la connexion. Cela permet aux utilisateurs de ne pas s'authentifier manuellement à chaque fois qu'ils accèdent à notre site. Ajoutez donc l'option `remember_me` dans la configuration. Voici ce que cela donne :

Code : YAML

```

# app/config/security.yml

security:
    # ...
    firewalls:
        # ... le pare-feu « dev »
        # Firewall principal pour le reste de notre site
        main:
            pattern:      ^
            anonymous:   true
            provider:    main
            form_login:
                login_path: login
                check_path: login_check
            logout:
                path:       logout
                target:    /blog
            remember_me:
                key:        %secret% # %secret% est un paramètre de
parameters.yml

```

J'ai juste ajouté le dernier paramètre `remember_me`.

Configuration de la sécurité : check !

Et voilà, votre site est prêt à être sécurisé ! En effet, on a fini de configurer la sécurité pour utiliser tout ce qu'offre le bundle à ce niveau.

Pour tester à nouveau si tout fonctionne, il faut ajouter des utilisateurs à notre base de données. Pour cela, on ne va pas réutiliser nos *fixtures* précédentes, mais on va utiliser une commande très sympa proposée par FOSUserBundle. Exécutez la commande suivante et laissez-vous guider :

Code : Console

```
php app/console fos:user:create
```

Vous l'aurez deviné, c'est une commande très pratique qui permet de créer des utilisateurs facilement. Laissez-vous guider, elle vous demande le nom d'utilisateur, l'e-mail et le mot de passe, et hop ! elle crée l'utilisateur. Vous pouvez aller vérifier le résultat dans phpMyAdmin. Notez au passage que le mot de passe a bien été encodé, en sha512 comme on l'a demandé.

FOSUserBundle offre bien plus que seulement de la sécurité. Du coup, maintenant que la sécurité est bien configurée, passons au reste de la configuration du bundle.

Configuration du bundle FOSUserBundle

Configuration des routes

En plus de gérer la sécurité, le bundle FOSUserBundle gère aussi les pages classiques comme la page de connexion, celle d'inscription, etc. Pour toutes ces pages, il faut évidemment enregistrer les routes correspondantes. Les développeurs du bundle ont volontairement éclaté toutes les routes dans plusieurs fichiers pour pouvoir personnaliser facilement toutes ces pages. Pour l'instant, on veut juste les rendre disponibles, on les personnalisera plus tard. Ajoutez donc dans votre `routing.yml` les imports suivants à la suite du nôtre :

Code : YAML

```
# app/config/routing.yml
#
fos_user_security:
    resource: "@FOSUserBundle/Resources/config/routing/security.xml"

fos_user_profile:
    resource: "@FOSUserBundle/Resources/config/routing/profile.xml"
    prefix: /profile

fos_user_register:
    resource:
        "@FOSUserBundle/Resources/config/routing/registration.xml"
    prefix: /register

fos_user_resetting:
    resource:
        "@FOSUserBundle/Resources/config/routing/resetting.xml"
    prefix: /resetting

fos_user_change_password:
    resource:
        "@FOSUserBundle/Resources/config/routing/change_password.xml"
    prefix: /profile
```

Vous remarquez que les routes sont définies en XML et non en YML comme on en a l'habitude dans ce cours. En effet, je vous en avais parlé tout au début, Symfony2 permet d'utiliser plusieurs méthodes pour les fichiers de configuration : YML, XML et même PHP, au choix du développeur. Ouvrez ces fichiers de routes pour voir à quoi ressemblent des routes en XML. C'est quand même moins lisible qu'en YML, c'est pour cela qu'on a choisi YML au début. 😊

Ouvrez vraiment ces fichiers pour connaître toutes les routes qu'ils contiennent. Vous saurez ainsi faire des liens vers toutes les pages qu'offre le bundle : inscription, mot de passe perdu, etc. Inutile de réinventer la roue ! Voici quand même un extrait de la commande `php app/console router:debug` pour les routes qui concernent ce bundle :

Code : Autre

<code>fos_user_security_login</code>	ANY	ANY	/login
<code>fos_user_security_check</code>	ANY	ANY	/login_check
<code>fos_user_security_logout</code>	ANY	ANY	/logout
<code>fos_user_profile_show</code>	GET	ANY	/profile/
<code>fos_user_profile_edit</code>	ANY	ANY	/profile/edit

fos_user_registration_register	ANY	ANY	/register/
fos_user_registration_check_email	GET	ANY	/register/check-email
fos_user_registration_confirm	GET	ANY	/register/confirm/{token}
fos_user_registration_confirmed	GET	ANY	/register/confirmed
fos_user_resetting_request	GET	ANY	/resetting/request
fos_user_resetting_send_email	POST	ANY	/resetting/send-email
fos_user_resetting_check_email	GET	ANY	/resetting/check-email
fos_user_resetting_reset	GET POST	ANY	/resetting/reset/{token}
fos_user_change_password	GET POST	ANY	/profile/change-password

Vous notez que le bundle définit également les routes de sécurité /login et autres. Du coup, je vous propose de laisser le bundle gérer cela, supprimez donc les trois routes login, login_check et logout qu'on avait déjà définies et qui ne servent plus. De plus, il faut adapter la configuration du pare-feu, car le nom de ces routes a changé, voici ce que cela donne :

Code : YAML

```
# app/config/security.yml

security:
    firewalls:
        main:
            pattern:      ^
            anonymous:   true
            provider:    main
            form_login:
                login_path: fos_user_security_login
                check_path:  fos_user_security_check
            logout:
                path:        fos_user_security_logout
                target:      /blog
            remember_me:
                key:         %secret% # %secret% est un paramètre de
parameters.yml
```

 Comme notre bundle SdzUserBundle hérite de FOSUserBundle, c'est notre contrôleur et donc notre vue qui sont utilisés sur la route login pour l'instant, car les noms que nous avions utilisés sont les mêmes que ceux de FOSUserBundle. Étant donné que le contrôleur de FOSUserBundle apporte un petit plus (protection CSRF notamment), je vous propose de supprimer notre contrôleur SecurityController et notre vue Security/login.html.twig pour laisser ceux de FOSUserBundle prendre la main.

Il reste quelques petits détails à gérer comme la page de login qui n'est pas la plus sexy, sa traduction, et aussi un bouton « Déconnexion », parce que changer manuellement l'adresse en /logout, c'est pas super *user-friendly* !

Personnalisation esthétique du bundle

Heureusement tout cela est assez simple.

 Attention, la personnalisation esthétique que nous allons faire ne concerne en rien la couche sécurité à proprement parler. Soyez bien conscients de la différence !

Intégrer les pages du bundle dans notre layout

FOSUserBundle utilise un layout volontairement simpliste, parce qu'il a vocation à être remplacé par le nôtre. Le layout actuel est le suivant : [https://github.com/FriendsOfSymfony/FO \[...\] out.html.twig](https://github.com/FriendsOfSymfony/FO [...] out.html.twig)

On va donc tout simplement le remplacer par une vue Twig qui va étendre notre layout (qui est dans

app/Resources/views/layout.html.twig, rappelez-vous). Pour « remplacer » le layout du bundle, on va utiliser l'un des avantages d'avoir hérité de ce bundle dans le nôtre, en créant une vue du même nom dans notre bundle. Créez donc la vue layout.html.twig suivante :

Code : HTML & Django

```
{# src/Sdz/UserBundle/Resources/views/layout.html.twig #-}

{# On étend notre layout #-}
{%- extends "::layout.html.twig" %}

{# Dans notre layout, il faut définir le block body #-}
{%- block body %}

    {# On affiche les messages flash que définissent les contrôleurs
       du bundle #}
    {%- for key, message in app.session.flashbag.all() %}
        <div class="alert alert-{{ key }}">
            {{ message|trans({}, 'FOSUserBundle') }}
        </div>
    {%- endfor %}

    {# On définit ce block, dans lequel vont venir s'insérer les
       autres vues du bundle #}
    {%- block fos_user_content %}{% endblock fos_user_content %}

{%- endblock %}
```



Pour créer ce layout je me suis simplement inspiré de celui fourni par FOSUserBundle, en l'adaptant juste à notre cas.

Et voilà, si vous actualisez la page [/login](#) (après vous être déconnectés via [/logout](#) évidemment), vous verrez que le formulaire de connexion est parfaitement intégré dans notre design ! Vous pouvez également tester la page d'inscription sur [/register](#), qui est bien intégrée aussi.



Votre layout n'est pas pris en compte ? N'oubliez jamais d'exécuter la commande
`php app/console cache:clear` lorsque vous avez des erreurs qui vous étonnent !

Traduire les messages

FOSUB étant un bundle international, le texte est géré par le composant de traduction de Symfony2. Par défaut, celui-ci est désactivé. Pour traduire le texte il suffit donc d'activer (direction le fichier config.yml) et de décommenter une des premières lignes dans framework :

Code : YAML

```
# app/config/config.yml

framework:
    translator: { fallback: %locale% }
```

Où %locale% est un paramètre défini dans app/config/parameters.yml, et que vous pouvez mettre à « fr » si ce n'est pas déjà fait. Ainsi, tous les messages utilisés par FOSUserBundle seront traduits en français !

Afficher une barre utilisateur

Il est intéressant d'afficher dans le layout si le visiteur est connecté ou non, et d'afficher des liens vers les pages de connexion ou de déconnexion. Cela se fait facilement, je vous invite à insérer ceci dans votre layout, où vous voulez :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #}

{%
    if is_granted("IS_AUTHENTICATED_REMEMBERED") %}
        Connecté en tant que {{ app.user.username }} - <a href="{{ path('fos_user_security_logout') }}>Déconnexion</a>
{%
    else %}
        <a href="{{ path('fos_user_security_login') }}>Connexion</a>
{%
    endif %}
```

Adaptez et mettez ce code dans votre layout, effet garanti. 😊

 Le rôle `IS_AUTHENTICATED_REMEMBERED` est donné à un utilisateur qui s'est authentifié soit automatiquement grâce au cookie `remember_me`, soit en utilisant le formulaire de connexion. Le rôle `IS_AUTHENTICATED_FULLY` est donné à un utilisateur qui s'est obligatoirement authentifié manuellement, en rentrant son mot de passe dans le formulaire de connexion. C'est utile pour protéger les opérations sensibles comme le changement de mot de passe ou d'adresse e-mail.

Manipuler les utilisateurs avec FOSUserBundle

Nous allons voir les moyens pour manipuler vos utilisateurs au quotidien.

Si les utilisateurs sont gérés par FOSUserBundle, ils ne restent que des entités Doctrine2 des plus classiques. Ainsi, vous pourriez très bien vous créer un repository comme vous savez le faire. Cependant, profitons du fait que le bundle intègre un `UserManager` (c'est une sorte de repository avancé). Ainsi, voici les principales manipulations que vous pouvez faire avec :

Code : PHP

```
<?php
// Dans un contrôleur :

// Pour récupérer le service UserManager du bundle
$userManager = $this->get('fos_user.user_manager');

// Pour charger un utilisateur
$user = $userManager->findUserBy(array('username' => 'winzou'));

// Pour modifier un utilisateur
$user->setEmail('cetemail@nexiste.pas');
$userManager->updateUser($user); // Pas besoin de faire un flush avec l'EntityManager, cette méthode le fait toute seule !

// Pour supprimer un utilisateur
$userManager->deleteUser($user);

// Pour récupérer la liste de tous les utilisateurs
$users = $userManager->findUsers();
```

Si vous avez besoin de plus de fonctions, vous pouvez parfaitement faire un repository personnel, et le récupérer comme d'habitude via `$this->getDoctrine()->getManager()->getRepository('SdzUserBundle:User')`. Et si vous voulez en savoir plus sur ce que fait le bundle dans les coulisses, n'hésitez pas à aller voir le code des contrôleurs du bundle.

Pour conclure

Ce chapitre touche à sa fin. Vous avez maintenant tous les outils en main pour construire votre espace membres, avec un système d'authentification performant et sécurisé, et des accès limités pour vos pages suivant des droits précis.

Sachez que tout ceci n'est qu'une introduction à la sécurité sous Symfony2. Les processus complets sont très puissants mais évidemment plus complexes. Si vous souhaitez aller plus loin pour faire des opérations plus précises (authentification Facebook, LDAP, etc.), n'hésitez pas à vous référer à [la documentation officielle sur la sécurité](#). Allez jeter un œil également à [la documentation de FOSUserBundle](#), qui explique comment personnaliser au maximum le bundle, ainsi que l'[utilisation des groupes](#).

Pour information, il existe également un système d'ACL, qui vous permet de définir des droits bien plus finement que les rôles. Par exemple, pour autoriser l'édition d'un article si on est admin *ou* si on en est l'auteur. Je ne traiterai pas ce point dans ce cours, mais n'hésitez pas à vous référer à [la documentation à ce sujet](#).

- La sécurité se compose de deux couches :
 - L'authentification, qui définit qui est le visiteur ;
 - L'autorisation, qui définit si le visiteur a accès à la ressource demandée.
- Le fichier `security.yml` permet de configurer finement chaque acteur de la sécurité :
 - La configuration de l'authentification passe surtout par le paramétrage d'un ou plusieurs pare-feu ;
 - La configuration de l'autorisation se fait au cas par cas suivant les ressources : on peut sécuriser une méthode de contrôleur, un affichage ou une URL.
- Les rôles associés aux utilisateurs définissent les droits dont ils disposent ;
- On peut configurer la sécurité pour utiliser `FOSUserBundle`, un bundle qui offre un espace membres presque clé en main.

Les services, utilisation poussée

Ce chapitre fait suite au précédent chapitre sur les services, qui portait sur la théorie et l'utilisation simple.

Ici nous allons aborder des fonctionnalités intéressantes des services, qui permettent une utilisation vraiment poussée. Maintenant que les bases vous sont acquises, nous allons pouvoir découvrir des fonctionnalités très puissantes de Symfony.

Les tags sur les services

Les tags ?

Une fonctionnalité très importante des services est la possibilité d'utiliser les **tags**. Un tag est une option qu'on appose à un ou plusieurs services afin que le conteneur de services les identifie comme tels. Ainsi, il devient possible de récupérer tous les services qui possèdent un certain tag.

C'est un mécanisme très pratique pour ajouter des fonctionnalités à un composant. Pour bien comprendre, prenons l'exemple de Twig.

Comprendre les tags à travers Twig

Le moteur de templates Twig dispose nativement de plusieurs fonctions pratiques pour vos vues. Seulement, il serait intéressant de pouvoir ajouter nos propres fonctions qu'on pourra utiliser dans nos vues. Vous l'aurez compris, c'est possible grâce au mécanisme des tags.

Appliquer un tag à un service

Pour que Twig puisse récupérer tous les services qui vont définir des fonctions supplémentaires utilisables dans nos vues, il faut appliquer un tag à ces services.

Si vous avez toujours le service `SdzAntispam` défini au précédent chapitre sur les services, je vous invite à rajouter le tag dans la configuration du service. Bien entendu, si vous n'aviez pas ce service, rien ne vous empêche d'en créer un nouveau : vous taguez les services que vous voulez.

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdz_blog.antispam:
        class:      Sdz\BlogBundle\Antispam\SdzAntispam
        arguments:  [@mailer, %locale%, 3]
        tags:
            - { name: twig.extension }
```

Vous voyez, on a simplement rajouté un attribut `tags` à la configuration de notre service. Cet attribut contient un tableau de tags, d'où le retour à la ligne et le tiret « - ». En effet, il est tout à fait possible d'associer plusieurs tags à un même service.

Dans ce tableau de tags, on a ajouté une ligne avec un attribut `name`, qui est le nom du tag. C'est grâce à ce nom que Twig va pouvoir récupérer tous les services avec ce tag. Ici `twig.extension` est donc le tag qu'on utilise.

Bien entendu, maintenant que Twig peut récupérer tous les services tagués, il faut que tout le monde puisse se comprendre.

Une classe qui implémente une interface

Celui qui va récupérer les services d'un certain tag attend un certain comportement de la part des services qui ont ce tag. Il faut donc les faire implémenter une interface ou étendre une classe de base.

En particulier, Twig attend que votre service implémente l'interface `Twig_ExtensionInterface`. Encore plus simple, Twig propose une classe abstraite à hériter par notre service, il s'agit de `Twig_Extension`. Je vous invite donc à modifier notre classe `SdzAntispam` pour qu'elle hérite de `Twig_Extension` (qui, elle, implémente `Twig_ExtensionInterface`) :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Antispam/SdzAntispam.php

namespace Sdz\BlogBundle\Antispam;

class SdzAntispam extends \Twig_Extension
{
    // ...
}
```

Notre service est prêt à fonctionner avec Twig, il ne reste plus qu'à écrire au moins une des méthodes de la classe abstraite `Twig_Extension`.

Écrire le code qui sera exécuté

Cette section est propre à chaque tag, où celui qui récupère les services d'un certain tag va exécuter telle ou telle méthode des services. En l'occurrence, Twig va exécuter les méthodes suivantes :

- `getFilters()`, qui retourne un tableau contenant les filtres que le service ajoute à Twig ;
- `getTests()`, qui retourne les tests ;
- `getFunctions()`, qui retourne les fonctions ;
- `getOperators()`, qui retourne les opérateurs ;
- `getGlobals()`, qui retourne les variables globales.

Pour notre exemple, nous allons juste ajouter une fonction accessible dans nos vues via `{ { checkIfSpam('le message') } }`. Elle vérifie si son argument est un spam. Pour cela, écrivons la méthode `getFunctions()` suivante dans notre service :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Antispam/SdzAntispam.php

namespace Sdz\BlogBundle\Antispam;

class SdzAntispam extends \Twig_Extension
{
    /*
     * Twig va exécuter cette méthode pour savoir quelle(s) fonction(s)
     * ajoute notre service
    */
    public function getFunctions()
    {
        return array(
            'checkIfSpam' => new \Twig_Function_Method($this, 'isSpam')
        );
    }

    /*
     * La méthode getName() identifie votre extension Twig, elle est
     * obligatoire
    */
    public function getName()
    {
        return 'SdzAntispam';
    }

    // ...
}
```

Dans cette méthode `getFunctions()` :

- `checkIfSpam` est le nom de la fonction qui sera disponible sous Twig, via `{ { checkIfSpam(var) } }`.
- `new \Twig_Function_Method($this, 'isSpam')` est ce qui sera exécuté lors de l'appel à la fonction Twig `checkIfSpam()`. Ici, il s'agit de la méthode `isSpam()` de l'objet courant `$this`.
- Au final, `{ { checkIfSpam(var) } }` côté Twig exécute `$this->isSpam($var)` côté service.

On a également ajouté la méthode `getName()` qui identifie votre service de manière unique parmi les extensions Twig, elle est obligatoire, ne l'oubliez pas.

Et voilà ! Vous pouvez dès à présent utiliser la fonction `{ { checkIfSpam() } }` dans vos vues. 😊

Méthodologie

Ce qu'on vient de faire pour transformer notre simple service en extension Twig est la méthodologie à appliquer systématiquement lorsque vous taguez un service. Sachez que tous les tags ne nécessitent pas forcément que votre service implémente une certaine interface, mais c'est assez fréquent.

Pour connaître tous les services implémentant un certain tag, vous pouvez exécuter la commande suivante :

Code : Console

```
C:\wamp\www\Symfony> php app/console container:debug --tag=twig.extension
[container] Public services with tag twig.extension
Service Id          Scope      Class Name
sdz_blog.antispam   container  Sdz\BlogBundle\Antispam\SdzAntispam
twig.extension.intl  container  Twig_Extensions_Extension_Intl
```



Vous trouverez plus d'informations sur la création d'extensions Twig dans [la documentation de Twig](#).

Les principaux tags

Il existe pas mal de tags prédéfinis dans Symfony2, qui permettent d'ajouter des fonctionnalités à droite et à gauche. Je ne vais vous présenter ici que deux des principaux tags. Mais sachez que l'ensemble des tags est expliqué [dans la documentation](#).

Les évènements du cœur

Les services peuvent être utilisés avec le gestionnaire d'évènements, via plusieurs tags. Dans ce cas, les différents tags permettent d'exécuter un service à des moments précis dans l'exécution d'une page. Le gestionnaire d'évènements est un composant très intéressant, et fait l'objet d'un [prochain chapitre dédié](#).

Les types de champ de formulaire

Le tag `form.type` permet de définir un nouveau type de champ de formulaire. Par exemple, si vous souhaitez utiliser l'éditeur WYSIWYG (*What you see is what you get*) `ckeditor` pour certains de vos champs texte, il est facile de créer un champ `ckeditor` au lieu de `textarea`. Pour cela, disons que vous avez ajouté le JavaScript nécessaire pour activer cet éditeur sur les `<textarea>` qui possèdent la classe `ckeditor`. Il ne reste plus qu'à automatiser l'apparition de cette classe.

Commençons par créer la classe du type de champ :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Form/Type/CkeditorType.php

namespace Sdz\BlogBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class CkeditorType extends AbstractType
{
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'attr' => array('class' => 'ckeditor')
        ));
    }

    public function getParent()
    {
        return 'textarea';
    }

    public function getName()
    {
        return 'ckeditor';
    }
}

```

Ce type de champ hérite de toutes les fonctionnalités d'un `textarea` (grâce à la méthode `getParent()`) tout en disposant de la classe CSS `ckeditor` (définie dans la méthode `setDefaultOptions()`) vous permettant, en ajoutant `ckeditor` à votre site, de transformer vos `<textarea>` en éditeur WYSIWYG.

Puis, déclarons cette classe en tant que service, en lui ajoutant le tag `form.type` :

Code : YAML

```

# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdz_blog_ckeditor:
        class: Sdz\BlogBundle\Form\Type\CkeditorType
        tags:
            - { name: form.type, alias: ckeditor }

```

On a ajouté l'attribut `alias` dans le tag, qui représente le nom sous lequel on pourra utiliser ce nouveau type. Pour l'utiliser, c'est très simple, modifiez vos formulaires pour utiliser `ckeditor` à la place de `textarea`. Par exemple, dans notre `ArticleType` :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Form/ArticleType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ArticleType extends AbstractType

```

```
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
            // ...
            ->add('contenu', 'ckeditor')
        ;
    }

    // ...
}
```

Et voilà, votre champ a maintenant automatiquement la classe CSS `ckeditor`, ce qui permet d'activer l'éditeur (si vous l'avez ajouté à votre site bien sûr).

C'était un exemple pour vous montrer comment utiliser les tags dans ce contexte.



Pour plus d'informations sur la création de type de champ, je vous invite à lire la documentation à ce sujet.

Dépendances optionnelles : les calls

Les dépendances optionnelles

L'injection de dépendances dans le constructeur, comme on l'a fait dans le précédent chapitre sur les services, est un très bon moyen de s'assurer que la dépendance sera bien disponible. Mais parfois vous pouvez avoir des dépendances optionnelles. Ce sont des dépendances qui peuvent être rajoutées au milieu de l'exécution de la page, grâce à des setters. Reprenons par exemple notre service d'antispam, où l'on définit l'argument `$locale` comme optionnel. L'idée est de supprimer ce dernier des arguments du constructeur, et d'ajouter le setter correspondant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Antispam/SdzAntispam.php

namespace Sdz\BlogBundle\Antispam;

class SdzAntispam extends \Twig_Extension
{
    protected $mailer;
    protected $locale;
    protected $nbForSpam;

    // Dans le constructeur, on retire $locale des arguments
    public function __construct(\Swift_Mailer $mailer, $nbForSpam)
    {
        $this->mailer = $mailer;
        $this->nbForSpam = (int) $nbForSpam;
    }

    // Et on ajoute un setter
    public function setLocale($locale)
    {
        $this->locale = $locale;
    }

    // ...
}
```



N'oubliez pas de supprimer l'argument `%locale%` de la définition du service. Rappelez-vous : si vous modifiez le constructeur du service, vous devez adapter sa configuration, et inversement.

Cependant, dans ce cas, la dépendance optionnelle `$locale` n'est jamais renseignée par le conteneur, pas même avec une valeur par défaut. C'est ici que les **calls** interviennent.

Les calls

Les calls sont un moyen d'exécuter des méthodes de votre service juste après sa création. Ainsi, on peut exécuter la méthode `setLocale()` avec notre paramètre `%locale%`, qui sera une valeur par défaut pour ce service. Elle pourra tout à fait être écrasée par une autre au cours de l'exécution de la page.

Je vous invite donc à rajouter l'attribut `calls` à la définition du service, comme ceci :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdz_blog_antispam:
        class:      Sdz\BlogBundle\Antispam\SdzAntispam
        arguments:  [@doctrine, 3]
        calls:
            - [ setLocale, [ %locale% ] ]
```

Comme avec les tags, vous pouvez définir plusieurs calls, en rajoutant des lignes à tiret. Chaque ligne de call est un tableau qui se décompose comme suit :

- Le premier index, ici `setLocale`, est le nom de la méthode à exécuter ;
- Le deuxième index, ici `[%locale%]`, est le tableau des arguments à transmettre à la méthode exécutée. Ici nous avons un seul argument, mais il est tout à fait possible d'en définir plusieurs.

Concrètement, dans notre cas le code équivalent du conteneur serait celui-ci :

Code : PHP

```
<?php

$antispam = new \Sdz\BlogBundle\Antispam\SdzAntispam($doctrine, 3);
$antispam->setLocale($locale);
```



Ce code n'existe pas, c'est un code fictif pour vous représenter l'équivalent de ce que fait le conteneur de services dans son coin. 😊

L'utilité des calls

En plus du principe de dépendance optionnelle, l'utilité des calls est également remarquable pour l'intégration des bibliothèques externes (Zend Framework, GeSHI, etc.), qui ont besoin d'exécuter quelques méthodes en plus du constructeur. Vous pourrez donc le faire grâce aux calls.

Les champs d'application, ou scopes

Cette section traite des champs d'application des services, *scopes* en anglais. C'est une notion avancée et il faut que vous sachiez qu'elle existe. En effet, vous rencontrerez sûrement un jour une erreur mentionnant le scope d'un de vos services. Voici les clés pour vous en sortir.

La problématique

Pour bien comprendre la problématique posée par le scope d'un service, je vous propose un exemple simple.

Prenez notre service `Antispam`, pour lequel nous n'avons pas précisé de scope particulier. Le comportement par défaut du conteneur, comme je vous l'ai expliqué, est qu'à chaque fois que vous appelez le service `Antispam`, vous recevez le même objet. Il est en effet stocké dans le conteneur au premier appel, puis c'est le même objet qui vous est retourné toutes les fois suivantes.

Imaginons maintenant qu'on injecte la requête, le service `request`, dans notre service `Antispam`. On s'en sert pour récupérer l'URL courante par exemple, et faire un comportement différent selon sa valeur.

Voyons alors quelles sont les conséquences dans l'utilisation pratique de notre service `Antispam` :

1. Vousappelez le service `Antispam`. Une instance de la classe est donc créée, appelons-la `Antispam1`, avec comme argument une instance de la classe `Request`, appelons-la `Request1`.
2. Maintenant, vous effectuez une sous-requête, un `{% render %}` depuis une vue par exemple. C'est une sous-requête, donc le noyau de Symfony2 va créer une nouvelle requête, une nouvelle instance de `Request`, appelons-la `Request2`.
3. Dans cette sous-requête, vous faites encore appel au service `Antispam`. Rappelez-vous, c'est la même instance qui vous est retournée à chaque fois, vous utilisez donc à nouveau `Antispam1`. Or, cette instance `Antispam1` contient l'objet `Request1` et non `Request2` ! Du coup, l'URL que vous utilisez dans le service est ici erronée, puisque la requête a été changée entre temps !

Les scopes sont justement là pour éviter ce genre de problème. En effet, il est en réalité impossible d'être confronté à l'exemple que nous venons de voir, car le conteneur vous retournera une erreur si vous essayez d'injecter la requête dans un service dont le scope n'est pas défini.

Que sont les scopes ?

Le scope d'un service sert de cadre pour les interactions entre une instance de ce service et le conteneur de services. Le conteneur de Symfony2 offre trois scopes par défaut :

- `container` (la valeur par défaut) : le conteneur vous retourne la même instance du service à chaque fois que vous lui demandez ce service. C'est le comportement par défaut dont j'ai toujours parlé jusqu'à maintenant.
- `request` : le conteneur vous retourne la même instance du service au sein de la même requête. En cas de sous-requête, le conteneur crée une nouvelle instance du service et vous retourne la nouvelle, l'ancienne n'est plus accessible.
- `prototype` : le conteneur vous retourne une nouvelle instance à chaque fois que vous lui demandez ce service.

Les scopes induisent donc une contrainte sur les dépendances d'un service (ce que vous injectez dans ce service) : un service ne peut pas dépendre d'un autre service au scope plus restreint. Le scope `prototype` est plus restreint que `request`, qui lui-même est plus restreint que `container`.

Pour reprendre l'exemple précédent, si vous avez un service `SdzAntispam` générique (c'est-à-dire avec le scope `container` par défaut), mais que vous essayez de lui injecter le service `request` qui est de scope plus restreint, vous recevrez une erreur (`ScopeWideningInjectionException`) au moment de la compilation du conteneur.

Et concrètement ?

En pratique, vous rencontrerez cette contrainte sur les scopes à chaque fois que vous injecterez la requête dans un de vos services. Pensez donc, à chaque fois que vous le faites, à définir le scope à `request` dans la configuration de votre service :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdz_blog.antispam:
        class:      Sdz\BlogBundle\Antispam\SdzAntispam
        arguments:  [@request, @unAutreService]
        scope:     request
```

Et bien entendu, maintenant que ce service a le scope `request`, vous devrez appliquer ce même scope à tous les autres services dans lesquels vous l'injecterez.

Les services courants de Symfony2

Les services courants de Symfony

Maintenant que vous savez bien utiliser les services, il vous faut les connaître tous afin d'utiliser la puissance de chacun. Il est important de bien savoir quels sont les services existants afin de bien pouvoir injecter ceux qu'il faut dans les services que vous êtes amenés à créer de votre côté.

Je vous propose donc une liste des services par défaut de Symfony2 les plus utilisés. Gardez-la en tête !

Identifiant	Description
<code>doctrine.orm.entity_manager</code>	Ce service est l'instance de l' <code>EntityManager</code> de Doctrine ORM. On l'a rapidement évoqué dans la partie sur Doctrine, l' <code>EntityManager</code> est bien enregistré en tant que service dans Symfony2. Ainsi, lorsque dans un contrôleur vous faites <code>\$this->getDoctrine()->getManager()</code> , vous récupérez en réalité le service <code>doctrine.orm.entity_manager</code> . Ayez bien ce nom en tête, car vous aurez très souvent besoin de l'injecter dans vos propres services : il vous offre l'accès à la base de données, ce n'est pas rien !
<code>event_dispatcher</code>	Ce service donne accès au gestionnaire d'évènements. Le décrire en quelques lignes serait trop réducteur, je vous propose donc d'être patients, car le prochain chapitre lui est entièrement dédié. 😊
<code>kernel</code>	Ce service vous donne accès au noyau de Symfony. Grâce à lui, vous pouvez localiser des bundles, récupérer le chemin de base du site, etc. Voyez le fichier <code>Kernel.php</code> pour connaître toutes les possibilités. Nous nous en servirons très peu en réalité.
<code>logger</code>	Ce service gère les logs de votre application. Grâce à lui, vous pouvez utiliser des fichiers de logs très simplement. Symfony utilise la classe <code>Monolog</code> par défaut pour gérer ses logs. La documentation à ce sujet vous expliquera comment vous en servir si vous avez besoin d'enregistrer des logs pour votre propre application ; c'est intéressant, n'hésitez pas à vous renseigner.
<code>mailer</code>	Ce service vous renvoie par défaut une instance de <code>Swift_Mailer</code> , une classe permettant d'envoyer des e-mails facilement. Encore une fois, la documentation de <code>SwiftMailer</code> et la documentation de son intégration dans Symfony2 vous seront d'une grande aide si vous souhaitez envoyer des e-mails.
<code>request</code>	Ce service est très important : il vous donne une instance de <code>Request</code> qui représente la requête du client. Rappelez-vous, on l'a déjà utilisé en récupérant cette dernière directement via <code>\$this->getRequest()</code> depuis un contrôleur. Je vous réfère au chapitre sur les contrôleurs, section <code>Request</code> pour plus d'informations sur comment récupérer la session, l'IP du visiteur, la méthode de la requête, etc.
<code>router</code>	Ce service vous donne accès au routeur (<code>Symfony\Component\Routing\Router</code>). On l'a déjà abordé dans le chapitre sur les routes. Rappelez-vous, on générait des routes à l'aide du raccourci du contrôleur <code>\$this->generateUrl()</code> . Sachez aujourd'hui que cette méthode exécute en réalité <code>\$this->container->get('router')->generate()</code> . On utilisait déjà des services sans le savoir !
<code>security.context</code>	Ce service permet de gérer l'authentification sur votre site internet. On l'utilise notamment pour récupérer l'utilisateur courant. Le raccourci du contrôleur <code>\$this->getUser()</code> exécute en réalité <code>\$this->container->get('security.context')->getToken()->getUser()</code> !
<code>service_container</code>	Ce service vous renvoie le conteneur de services lui-même. On ne l'utilise que très rarement, car, comme je vous l'ai déjà mentionné, il est bien plus propre de n'injecter que les services dont on a besoin, et non pas tout le conteneur. Mais dans certains cas il est nécessaire de s'en servir, sachez donc qu'il existe.

session	Ce service représente les sessions. Vöyez le fichier Symfony\Component\HttpFoundation\Session\Session.php pour en savoir plus. Sachez également que la session est accessible depuis la requête, en faisant <code>\$request->getSession()</code> , donc si vous injectez déjà la requête dans votre service, inutile d'injecter également la session.
twig	Ce service représente une instance de Twig_Environment . Il permet d'afficher ou de retourner une vue. Vous pouvez en savoir plus en lisant la documentation de Twig . Ce service peut être utile pour modifier l'environnement de Twig depuis l'extérieur (lui ajouter des extensions, etc.).
templating	Ce service représente le moteur de templates de Symfony2. Par défaut il s'agit de Twig, mais cela peut également être PHP ou tout autre moteur intégré dans un bundle tiers. Ce service montre l'intérêt de l'injection de dépendances : en injectant templating et non twig dans votre service, vous faites un code valide pour plusieurs moteurs de templates ! Et si l'utilisateur de votre bundle utilise un moteur de templates à lui, votre bundle continuera de fonctionner. Sachez également que le raccourci du contrôleur <code>\$this->render()</code> exécute en réalité <code>\$this->container->get('templating')->renderResponse()</code> .

En résumé

- Les tags permettent de récupérer tous les services qui remplissent une même fonction : cela ouvre les possibilités pour les extensions Twig, les évènements, etc.
- Les calls permettent les dépendances optionnelles, et facilitent l'intégration de bibliothèques tierces.
- Les scopes sont un point particulier à connaître pour maîtriser complètement le conteneur de services.
- Les principaux noms de services sont à connaître par cœur afin d'injecter ceux nécessaires dans les services que vous créerez !

Le gestionnaire d'évènements de Symfony2

N'avez-vous jamais rêvé d'exécuter un certain code à chaque page consultée de votre site internet ? D'enregistrer chaque connexion des utilisateurs dans une base de données ? De modifier la réponse retornée au visiteur selon certains critères ? Eh bien, les développeurs de Symfony2 ne se sont pas contentés d'en rêver, ils l'ont fait !

En effet, comment réaliseriez-vous les cas précédents ? Avec un `if` sauvagement placé au fin fond d'un fichier ? Allons, un peu de sérieux, avec Symfony2 on va passer à la vitesse supérieure, et utiliser ce qu'on appelle le gestionnaire d'évènements.

Des évènements ? Pour quoi faire ?

Qu'est-ce qu'un évènement ?

Un évènement correspond à un moment clé dans l'exécution d'une page. Il en existe plusieurs, par exemple l'évènement `kernel.request` qui est déclenché avant que le contrôleur ne soit exécuté. Cet évènement est déclenché à chaque page, mais il en existe d'autres qui ne le sont que lors d'actions particulières, par exemple l'évènement `security.interactive_login`, qui correspond à l'identification d'un utilisateur.

Tous les évènements sont déclenchés à des endroits stratégiques de l'exécution d'une page Symfony2, et vont nous permettre de réaliser nos rêves de façon classe et surtout découpée.

Si vous avez déjà fait un peu de JavaScript, alors vous avez sûrement déjà traité des évènements. Par exemple, l'évènement `onClick` doit vous parler. Il s'agit d'un évènement qui est déclenché lorsque l'utilisateur clique quelque part, et on y associe une action (quelque chose à faire). Bien sûr, en PHP vous ne serez pas capables de détecter le clic utilisateur, c'est un langage serveur ! Mais l'idée est exactement la même.

Qu'est-ce que le gestionnaire d'évènements ?

J'ai parlé à l'instant de **découplage**. C'est la principale raison de l'utilisation d'un gestionnaire d'évènements ! Par code découpé, j'entends que celui qui écoute l'évènement ne dépend pas du tout de celui qui déclenche l'évènement. Je m'explique :

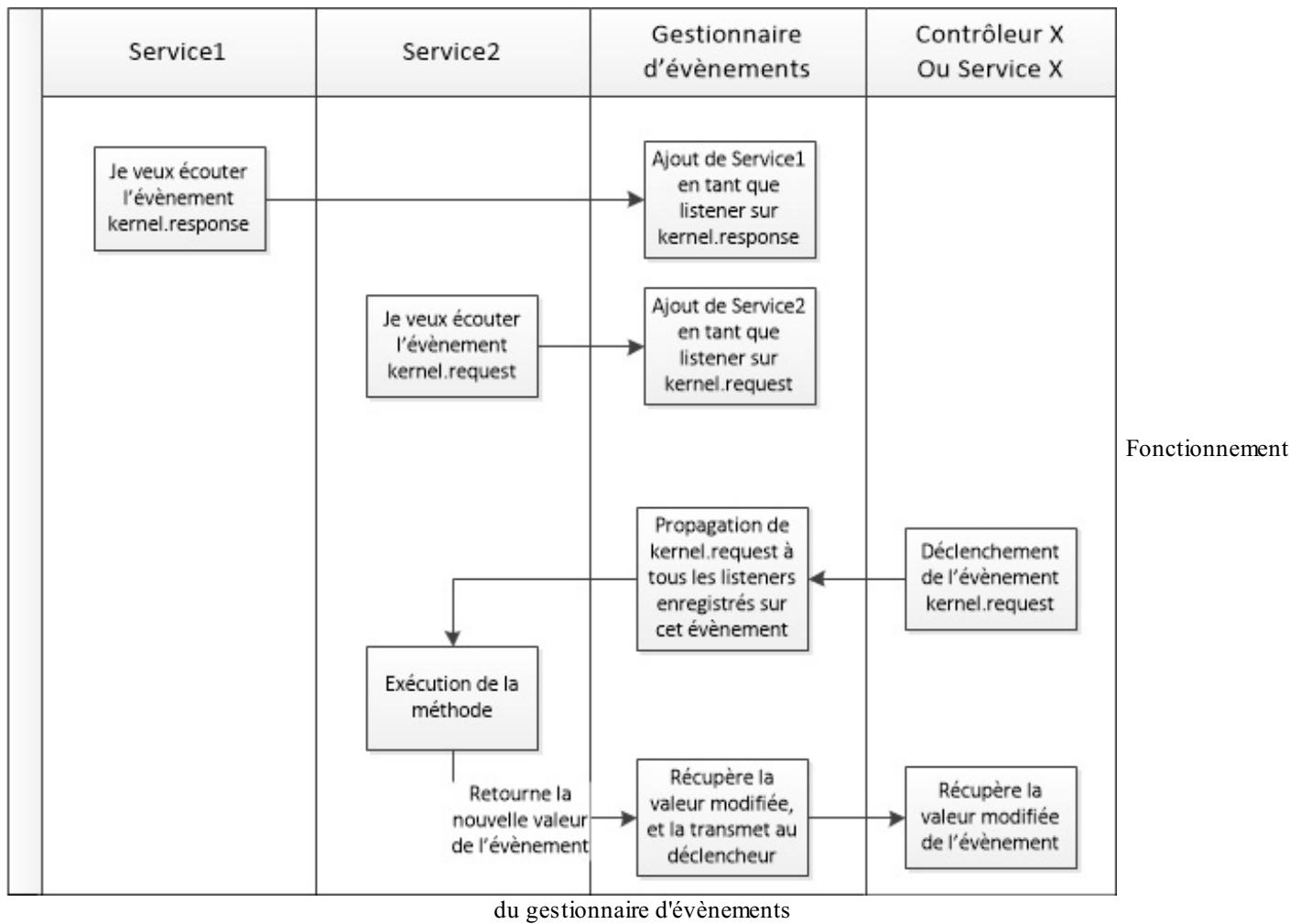
- On parle de **déclencher un évènement** lorsqu'on signale au gestionnaire d'évènements : « Tel évènement vient de se produire, préviens tout le monde, s'il-te-plaît. » Pour reprendre l'exemple de l'évènement `kernel.request`, c'est, vous l'aurez deviné, le Kernel qui déclenche l'évènement.
- On parle d'**écouter un évènement** lorsqu'on signale au gestionnaire d'évènements : « Je veux que tu me préviennes dès que tel évènement se produira, s'il-te-plaît. »

Ainsi, lorsqu'on écoute un évènement que le Kernel va déclencher, on ne touche pas au Kernel, on ne vient pas perturber son fonctionnement. On se contente d'exécuter du code de notre côté, en ne comptant que sur le déclenchement de l'évènement ; c'est le rôle du gestionnaire d'évènements de nous prévenir. Le code est donc totalement découpé, et en tant que bons développeurs, on aime ce genre de code !

Au niveau du vocabulaire, un service qui écoute un évènement s'appelle un **listener** (personne qui écoute, en français).

Pour bien comprendre le mécanisme, je vous propose un schéma sur la figure suivante montrant les deux étapes :

- Dans un premier temps, des services se font connaître du gestionnaire d'évènements pour écouter tel ou tel évènement. Ils deviennent des *listener* ;
- Dans un deuxième temps, quelqu'un (qui que ce soit) déclenche un évènement, c'est-à-dire qu'il prévient le gestionnaire d'évènements qu'un certain évènement vient de se produire. A partir de là, le gestionnaire d'évènement exécute chaque service qui s'est préalablement inscrit pour écouter cet évènement précis.



Certains d'entre vous auront peut-être reconnu le pattern Observer. En effet, le gestionnaire d'événements de Symfony2 n'est qu'une implémentation de ce pattern.

Écouter les événements

Notre exemple

Dans un premier temps, nous allons apprendre à écouter des événements. Pour cela je vais me servir d'un exemple simple : l'ajout d'une bannière « bêta » sur notre site, qui est encore en bêta car nous n'avons pas fini son développement ! L'objectif est donc de modifier chaque page retournée au visiteur pour ajouter cette balise.

L'exemple est simple, mais vous montre déjà le code découpé qu'il est possible de faire. En effet, pour afficher un « bêta » sur chaque page, il suffirait d'ajouter un ou plusieurs petits `if` dans la vue. Mais ce ne serait pas très joli, et le jour où votre site passe en stable il ne faudra pas oublier de retirer l'ensemble de ces `if`, bref, il y a un risque. Avec la technique d'un listener unique, il suffira de désactiver celui-ci.

Créer un listener

La première chose à faire est de créer le listener, car c'est lui qui va contenir la logique de ce qu'on veut exécuter. Une fois cela fait, il ne restera plus qu'à exécuter son code aux moments que l'on souhaite.

Pour savoir où placer le listener dans notre bundle, il faut se poser la question suivante : « À quelle fonction répond mon listener ? » La réponse est « À définir la version bêta », on va donc placer le listener dans le répertoire Beta, tout simplement. Pour l'instant il sera tout seul dans ce répertoire, mais si plus tard on doit rajouter un autre fichier qui concerne la même fonction, on le mettra avec.

Je vous invite donc à créer cette classe :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Beta/BetaListener.php

namespace Sdz\BlogBundle\Beta;
use Symfony\Component\HttpFoundation\Response;

class BetaListener
{
    // La date de fin de la version bêta :
    // - Avant cette date, on affichera un compte à rebours (J-3 par exemple)
    // - Après cette date, on n'affichera plus le « bêta »
    protected $dateFin;

    public function __construct($dateFin)
    {
        $this->dateFin = new \Datetime($dateFin);
    }

    // Méthode pour ajouter le « bêta » à une réponse
    protected function displayBeta(Response $reponse, $joursRestant)
    {
        $content = $reponse->getContent();

        // Code à rajouter
        $html = '<span style="color: red; font-size: 0.5em;"> - Beta J-' . (int) $joursRestant. '</span>';

        // Insertion du code dans la page, dans le <h1> du header
        $content = preg_replace('#<h1>(.*)</h1>#iu',
                               '<h1>$1'.$html.'</h1>',
                               $content,
                               1);

        // Modification du contenu dans la réponse
        $reponse->setContent($content);

        return $reponse;
    }
}

```

Pour l'instant, c'est une classe tout simple, qui n'écoute personne pour le moment. On dispose d'une méthode `displayBeta` prête à l'emploi pour modifier la réponse lorsqu'on la lui donnera.

Voici donc la base de tout listener : une classe capable de remplir une ou plusieurs fonctions. À nous ensuite d'exécuter la ou les méthodes aux moments opportuns.

Écouter un évènement

Vous le savez maintenant, pour que notre classe précédente écoute quelque chose, il faut la présenter au gestionnaire d'évènements. Il existe deux manières de le faire : manipuler directement le gestionnaire d'évènements, ou passer par les services.

Je ne vous cache pas qu'on utilisera très rarement la première méthode, mais je vais vous la présenter en premier, car elle permet de bien comprendre ce qu'il se passe dans la deuxième.



Vu que nous avons besoin de modifier la réponse retournée par les contrôleurs, nous allons écouter l'évènement `kernel.response`. Je vous dresse plus loin une liste complète des évènements avec les informations clés pour déterminer lequel écouter. Pour le moment, suivons notre exemple.

Méthode 1 : Manipuler directement le gestionnaire d'évènements

Cette première méthode, un peu brute, consiste à passer notre objet `BetaListener` au gestionnaire d'évènements. Ce

gestionnaire existe en tant que service sous Symfony, il s'agit de l'`EventDispatcher`. Plus simple, ça n'existe pas. Concrètement, voici comment faire :

Code : PHP

```
<?php
// Depuis un contrôleur

use Sdz\BlogBundle\Beta\BetaListener;

// ...

// On instancie notre listener
$betaListener = new BetaListener('2013-08-19');

// On récupère le gestionnaire d'événements, qui heureusement est
// un service !
$dispatcher = $this->get('event_dispatcher');

// On dit au gestionnaire d'exécuter la méthode onKernelResponse de
// notre listener
// Lorsque l'événement kernel.response est déclenché
$dispatcher->addListener('kernel.response', array($betaListener,
'onKernelResponse'));
```

À partir de maintenant, dès que l'événement `kernel.response` est déclenché, le gestionnaire d'événements exécutera `$betaListener->onKernelResponse()`. En effet, cette méthode n'existe pas encore dans notre listener, on en reparle dans quelques instants.



Bien évidemment, avec cette méthode, le moment où vous exécutez ce code est important ! En effet, si vous prévenez le gestionnaire d'événements *après* que l'événement qui vous intéresse s'est produit, votre listener ne sera pas exécuté !

Méthode 2 : Définir son listener comme service

Comme je vous l'ai dit, c'est cette méthode qu'on utilisera 99 % du temps. Elle est beaucoup plus simple et permet d'éviter le problème d'événement qui se produit avant l'enregistrement de votre listener dans le gestionnaire d'événements.

Mettons en place cette méthode pas à pas. Tout d'abord, définissez votre listener en tant que service, comme ceci :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdzblog.beta_listener:
        class: Sdz\BlogBundle\Beta\BetaListener
        arguments: ["2013-08-19"]
```

À partir de maintenant, votre listener est accessible via le conteneur de services. Pour aller plus loin, il faut définir le tag `kernel.event_listener` sur ce service. Le processus est le suivant : une fois le gestionnaire d'événements instancié par le conteneur de services, il va récupérer tous les services qui ont ce tag, et exécuter le code de la méthode 1 qu'on vient de voir afin d'enregistrer les listeners dans lui-même. Tout se fait automatiquement !

Voici donc le tag en question à rajouter à notre service :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml
```

```

services:
    sdzblog.beta_listener:
        class: Sdz\BlogBundle\Beta\BetaListener
        arguments: ["2013-08-19"]
        tags:
            - { name: kernel.event_listener, event: kernel.response,
method: onKernelResponse }

```

Il y a deux paramètres à définir dans le tag, qui sont les deux paramètres qu'on a utilisés précédemment dans la méthode `$dispatcher->addListener()` :

- `event` : c'est le nom de l'évènement que le listener veut écouter ;
- `method` : c'est le nom de la méthode du listener à exécuter lorsque l'évènement est déclenché.

C'est tout ! Avec uniquement cette définition de service et le bon tag associé, votre listener sera exécuté à chaque déclenchement de l'évènement `kernel.response` !

Bien entendu, votre listener peut tout à fait écouter plusieurs évènements. Il suffit pour cela d'ajouter un autre tag avec des paramètres différents. Voici ce que cela donnerait si on voulait écouter l'évènement `kernel.controller` :

Code : YAML

```

# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdzblog.beta_listener:
        class: Sdz\BlogBundle\Beta\BetaListener
        arguments: ["2013-08-19"]
        tags:
            - { name: kernel.event_listener, event: kernel.response,
method: onKernelResponse }
            - { name: kernel.event_listener, event:
kernel.controller, method: onKernelController }

```

Maintenant, passons à cette fameuse méthode `onKernelResponse`.

Création de la méthode à exécuter du listener



Mais pourquoi exécuter la méthode `onKernelResponse` alors qu'on avait déjà codé la méthode `displayBeta` ?

Réponse : par convention !

Plus sérieusement, dans votre esprit il faut bien distinguer deux points : d'un côté la fonction que remplit votre classe, et de l'autre la façon dont elle remplit sa fonction. Ici, notre classe `BetaListener` remplit parfaitement sa fonction avec la méthode `displayBeta`, c'est elle la méthode reine. Tout ce qu'on fait ensuite n'est qu'une solution technique pour arriver à notre but. En l'occurrence, le gestionnaire d'évènements ne va pas savoir donner les bons arguments (`$reponse` et `$joursRestant`) à notre méthode : il est alors hors de question de modifier notre méthode reine pour l'adapter au gestionnaire d'évènements !

En pratique, le gestionnaire donne un unique argument aux méthodes qu'il exécute pour nous : il s'agit d'un objet `Symfony\Component\EventDispatcher\Event`, représentant l'évènement en cours. Dans notre cas de l'évènement `kernel.response`, on a le droit à un objet `Symfony\Component\HttpKernel\Event\FilterResponseEvent`, qui hérite bien évidemment du premier.



Pas d'inquiétude : je vous dresse plus loin une liste des évènements Symfony2 ainsi que les types d'argument que le gestionnaire d'évènements transmet. Vous n'avez pas à jouer aux devinettes !

Notre évènement `FilterResponseEvent` dispose des méthodes suivantes :

Code : PHP

```
<?php
class FilterResponseEvent
{
    public function getResponse();
    public function setResponse(Response $response);
    public function getKernel();
    public function getRequest();
    public function getRequestType();
    public function isPropagationStopped();
    public function stopPropagation();
}
```

Dans notre cas, ce sont surtout les méthodes `getResponse()` et `setResponse` qui vont nous être utiles : elles permettent respectivement de récupérer la réponse et de la modifier, c'est exactement ce que l'on veut !

On a maintenant toutes les informations nécessaires, il est temps de construire la méthode `onKernelResponse` dans notre listener. Tout d'abord, voici le principe général pour ce type de listener qui vient *modifier* une partie de l'évènement (ici, la réponse) :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Beta/BetaListener.php

namespace Sdz\BlogBundle\Beta;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class BetaListener
{
    // ...

    public function onKernelResponse(FilterResponseEvent $event)
    {
        // On teste si la requête est bien la requête principale
        if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
            return;
        }

        // On récupère la réponse que le noyau a insérée dans
        // l'évènement
        $response = $event->getResponse();

        // Ici on modifie comme on veut la réponse...

        // Puis on insère la réponse modifiée dans l'évènement
        $event->setResponse($response);
    }
}
```

 Le premier `if` teste si la requête courante est bien la requête principale. En effet, souvenez-vous, on peut effectuer des sous-requêtes via la balise `{% render %}` de Twig ou alors la méthode `$this->forward()` d'un contrôleur. Cette condition permet de ne pas réexécuter le code lors d'une sous-requête (on ne va pas mettre des mentions « bête » sur chaque sous-requête !). Bien entendu, si vous souhaitez que votre comportement s'applique même aux sous-requêtes, ne mettez pas cette condition.

Adaptions maintenant cette base à notre exemple, il suffit juste de rajouter l'appel à notre précédente méthode, voyez par vous-mêmes :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Beta/BetaListener.php

namespace Sdz\BlogBundle\Beta;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class BetaListener
{
    // ...

    public function onKernelResponse(FilterResponseEvent $event)
    {
        // On teste si la requête est bien la requête principale
        if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
            return;
        }

        // On récupère la réponse depuis l'évènement
        $response = $event->getResponse();

        $joursRestant = $this->dateFin->diff(new \Datetime())->days;

        if ($joursRestant > 0) {
            // On utilise notre méthode « reine »
            $response = $this->displayBeta($event->getResponse(),
$joursRestant);
        }

        // On n'oublie pas d'enregistrer les modifications dans
        // l'évènement
        $event->setResponse($response);
    }
}
```



N'oubliez pas de rajouter le `use Symfony\Component\HttpKernel\Event\FilterResponseEvent;` en début de fichier.

Voilà, votre listener est maintenant opérationnel ! Actualisez n'importe quelle page de votre site et vous verrez la mention « bêta » apparaître comme sur la figure suivante.

The screenshot shows a website header with the text "Mon Projet Symfony2 - Beta J-209 !". Below the header, it says "Ce projet est propulsé par Symfony2, et construit grâce au tutoriel du siteduzero." and has a button labeled "Lire le tutoriel ➤". To the right, the text "La mention « bêta »" is written. The word "Beta" is highlighted in red, matching the color of the "Beta J-209 !" text in the header.

apparaît

Ici la bonne exécution de votre listener est évidente, car on a modifié l'affichage, mais parfois rien n'est moins sûr. Pour vérifier que votre listener est bien exécuté, allez dans l'onglet Events du Profiler, vous devez l'apercevoir dans le tableau visible à la figure suivante.

Called Listeners

Event name	Priority	Listener
kernel.request	1024	<code>ProfilerListener::onKernelRequest</code>
• • •		
kernel.response	0	<code>CacheListener::onKernelResponse</code>
kernel.response	0	<code>BetaListener::onKernelResponse</code>
kernel.response	0	<code>ResponseListener::onKernelResponse</code>

Notre listener figure dans la liste des listeners exécutés

Méthodologie

Vous connaissez maintenant la syntaxe pour créer un listener qui va écouter un évènement. Vous avez pu constater que le principe est assez simple, mais pour rappel voici la méthode à appliquer lorsque vous souhaitez écouter un évènement :

- Tout d'abord, créez une classe qui va remplir la fonction que vous souhaitez. Si vous avez un service déjà existant, cela va très bien.
- Ensuite, choisissez bien l'évènement que vous devez écouter. On a pris directement l'évènement `kernel.response` pour l'exemple, mais vous devez choisir correctement le vôtre dans la liste que je dresse plus loin.
- Puis créez la méthode qui va faire le lien entre le déclenchement de l'évènement et le code que vous voulez exécuter, il s'agit de la méthode `onKernelResponse` que nous avons utilisée.
- Enfin, définissez votre classe comme un service (sauf si c'en était déjà un), et ajoutez à la définition du service le bon tag pour que le gestionnaire d'évènements retrouve votre listener.

Les évènements Symfony2... et les nôtres !

Symfony2 déclenche déjà quelques évènements dans son processus interne. Mais il sera bien évidemment possible de créer puis déclencher nos propres évènements !

Les évènements Symfony2

L'évènement `kernel.request`

Cet évènement est déclenché très tôt dans l'exécution d'une page, avant même que le choix du contrôleur à exécuter ne soit fait. Son objectif est de permettre à un listener de retourner immédiatement une réponse, sans même passer par l'exécution d'un contrôleur donc. Il est également possible de définir des attributs dans la requête. Dans le cas où un listener définit une réponse, alors les listeners suivants ne seront pas exécutés ; on reparle de la priorité des listeners plus loin.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est `GetResponseEvent`, dont les méthodes sont les suivantes :

Code : PHP

```
<?php

class GetResponseEvent
{
    public function getResponse();
    public function setResponse(Response $response);
    public function hasResponse();
    public function getKernel();
    public function getRequest();
    public function getRequestType();
```

```
public function getDispatcher();
public function isPropagationStopped();
public function stopPropagation();
}
```

L'évènement kernel.controller

Cet évènement est déclenché après que le contrôleur à exécuter a été défini, mais avant de l'exécuter effectivement. Son objectif est de permettre à un listener de modifier le contrôleur à exécuter. Généralement, c'est également l'évènement utilisé pour exécuter du code sur chaque page.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est FilterControllerEvent, dont les méthodes sont les suivantes :

Code : PHP

```
<?php

class FilterControllerEvent
{
    public function getController();
    public function setController($controller);
    public function getKernel();
    public function getRequest();
    public function getRequestType();
    public function getDispatcher();
    public function isPropagationStopped();
    public function stopPropagation();
}
```

Voici comment utiliser cet évènement depuis un listener pour modifier le contrôleur à exécuter sur la page en cours :

Code : PHP

```
<?php

use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

public function onKernelController(FilterControllerEvent $event)
{
    // Vous pouvez récupérer le contrôleur que le noyau avait
    // l'intention d'exécuter
    $controller = $event->getController();

    // Ici vous pouvez modifier la variable $controller, etc.
    // $controller doit être de type PHP callable

    // Si vous avez modifié le contrôleur, prévenez le noyau qu'il
    // faut exécuter le vôtre :
    $event->setController($controller);
}
```

L'évènement kernel.view

Cet évènement est déclenché lorsqu'un contrôleur n'a pas retourné d'objet Response. Son objectif est de permettre à un listener d'attraper le retour du contrôleur (s'il y en a un) pour soit construire une réponse lui-même, soit personnaliser l'erreur levée.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est `GetResponseForControllerResultEvent` (rien que ça !), dont les méthodes sont les suivantes :

Code : PHP

```
<?php

class GetResponseForControllerResultEvent
{
    public function getControllerResult();
    public function getResponse();
    public function setResponse(Response $response);
    public function hasResponse();
    public function getKernel();
    public function getRequest();
    public function getRequestType();
    public function getDispatcher();
    public function isPropagationStopped();
    public function stopPropagation();
}
```

Voici comment utiliser cet évènement depuis un listener pour construire une réponse à partir du retour du contrôleur de la page en cours :

Code : PHP

```
<?php

use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelView(GetResponseForControllerResultEvent $event)
{
    // Récupérez le retour du contrôleur (ce qu'il a mis dans son « return »)
    $val = $event->getControllerResult();

    // Créez une nouvelle réponse
    $response = new Response();

    // Construisez votre réponse comme bon vous semble...

    // Définissez la réponse dans l'évènement, qui la donnera au noyau qui, finalement, l'affichera
    $event->setResponse($response);
}
```

L'évènement `kernel.response`

Cet évènement est déclenché après qu'un contrôleur a retourné un objet `Response` ; c'est celui que nous avons utilisé dans notre exemple de listener. Son objectif, comme vous avez pu vous en rendre compte, est de permettre à un listener de modifier la réponse générée par le contrôleur avant de l'envoyer à l'internaute.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est `FilterResponseEvent`, dont les méthodes sont les suivantes :

Code : PHP

```
<?php
```

```

class FilterResponseEvent
{
    public function getControllerResult();
    public function getResponse();
    public function setResponse(Response $response);
    public function hasResponse();
    public function getKernel();
    public function getRequest();
    public function getRequestType();
    public function getDispatcher();
    public function isPropagationStopped();
    public function stopPropagation();
}

```

L'évènement kernel.exception

Cet évènement est déclenché lorsqu'une exception est levée. Son objectif est de permettre à un listener de modifier la réponse à renvoyer à l'internaute, ou bien de modifier l'exception.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est GetResponseForExceptionEvent, dont les méthodes sont les suivantes :

Code : PHP

```

<?php

class GetResponseForExceptionEvent
{
    public function getException();
    public function setException(\Exception $exception);
    public function getResponse();
    public function setResponse(Response $response);
    public function hasResponse();
    public function getKernel();
    public function getRequest();
    public function getRequestType();
    public function getDispatcher();
    public function isPropagationStopped();
    public function stopPropagation();
}

```

L'évènement security.interactive_login

Cet évènement est déclenché lorsqu'un utilisateur s'identifie via le formulaire de connexion. Son objectif est de permettre à un listener d'archiver une trace de l'identification, par exemple.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est Symfony\Component\Security\Http\Event\InteractiveLoginEvent, dont les méthodes sont les suivantes :

Code : PHP

```

<?php

class InteractiveLoginEvent
{
    public function getAuthenticationToken();
    public function getRequest();
    public function getDispatcher();
    public function isPropagationStopped();
    public function stopPropagation();
}

```

```
}
```

L'évènement `security.authentication.success`

Cet évènement est déclenché lorsqu'un utilisateur s'identifie avec succès, quelque soit le moyen utilisé (formulaire de connexion, cookies `remember_me`). Son objectif est de permettre à un listener d'archiver une trace de l'identification, par exemple.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est `Symfony\Component\Security\Core\Event\AuthenticationEvent`, dont les méthodes sont les suivantes :

Code : PHP

```
<?php

class AuthenticationEvent
{
    public function getAuthenticationToken();
    public function getRequest();
    public function getDispatcher();
    public function isPropagationStopped();
    public function stopPropagation();
}
```

L'évènement `security.authentication.failure`

Cet évènement est déclenché lorsqu'un utilisateur effectue une tentative d'identification échouée, quelque soit le moyen utilisé (formulaire de connexion, cookies `remember_me`). Son objectif est de permettre à un listener d'archiver une trace de la mauvaise identification, par exemple.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est `Symfony\Component\Security\Core\Event\AuthenticationFailureEvent`, dont les méthodes sont les suivantes :

Code : PHP

```
<?php

class AuthenticationFailureEvent
{
    public function getAuthenticationException();
    public function getRequest();
    public function getDispatcher();
    public function isPropagationStopped();
    public function stopPropagation();
}
```



Notez la méthode `getDispatcher()` définie dans tous les évènements : elle récupère le gestionnaire d'évènements. Cela permet à un listener qui écoute un évènement A d'exécuter le code de la méthode n°1 vu précédemment dans sa méthode `onEvenementA`, il se mettra ainsi à écouter un évènement B sur sa méthode `onEvenementB`.

Créer nos propres évènements

Les évènements Symfony2 couvrent la majeure partie du *process* d'exécution d'une page, ou alors du *process* d'identification d'un utilisateur. Cependant, on aura parfois besoin d'appliquer cette conception par évènement à notre propre code, notre propre

logique. Cela permet encore une fois de bien découpler les différentes fonctions de notre site.

Nous allons suivre un autre exemple pour la création d'un évènement : celui d'un outil de surveillance des messages postés, qu'on appellera BigBrother. L'idée est d'avoir un outil qui permette de censurer les messages de certains utilisateurs et/ou de nous envoyer une notification lorsqu'ils postent des messages. L'avantage de passer par un évènement au lieu de modifier directement le contrôleur, c'est de pouvoir appliquer cet outil à plusieurs types de messages : les articles du blog, les commentaires du blog, les messages sur le forum si vous en avez un, etc.

Pour reproduire le comportement des évènements, il nous faut trois étapes :

- D'abord, définir la liste de nos évènements possibles. Il peut bien entendu y en avoir qu'un seul.
- Ensuite, construire la classe de l'évènement. Il faut pour cela définir les informations qui peuvent être échangées entre celui qui émet l'évènement et celui qui l'écoute.
- Enfin, déclencher l'évènement bien entendu.



Pour l'exemple, je vais placer tous les fichiers de la fonctionnalité BigBrother dans le répertoire `Bigbrother` du bundle Blog. Mais en réalité, comme c'est une fonctionnalité qui s'appliquera à plusieurs bundles (le blog, le forum, et d'autres si vous en avez), il faudrait le mettre dans un bundle séparé. Soit un bundle commun dans votre site, du genre `CoreBundle` si vous en avez un, soit carrément dans son bundle à lui, du genre `BigbrotherBundle`, vu que c'est une fonctionnalité que vous pouvez tout à fait partager avec d'autres sites !

Définir la liste de nos évènements

Nous allons définir une classe avec juste des constantes qui contiennent le nom de nos évènements. Cette classe est facultative en soi, mais c'est une bonne pratique qui nous évitera d'écrire directement le nom de l'évènement. On utilisera ainsi le nom de la constante, défini à un seul endroit, dans cette classe. J'appelle cette classe `BigbrotherEvents`, mais c'est totalement arbitraire, voici son code :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Bigbrother/BigbrotherEvents.php

namespace Sdz\BlogBundle\Bigbrother;

final class BigbrotherEvents
{
    const onMessagePost = 'sdzblog.bigbrother.post_message';
    // Vos autres évènements...
}
```

Cette classe ne fait donc rien, elle ne sert qu'à faire la correspondance entre `BigbrotherEvents::onMessagePost` qu'on utilisera pour déclencher l'évènement et le nom de l'évènement en lui même `sdzblog.bigbrother.post_message`.

Construire la classe de l'évènement

La classe de l'évènement, c'est, rappelez-vous, la classe de l'objet que le gestionnaire d'évènements va transmettre aux listeners. En réalité on ne l'a pas encore vu, mais c'est celui qui déclenche l'évènement qui crée une instance de cette classe. Le gestionnaire d'évènements ne fait que la transmettre, il ne la crée pas.

Voici dans un premier temps le squelette commun à tous les évènements. On va appeler le nôtre `MessagePostEvent` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Bigbrother/MessagePostEvent.php
```

```

namespace Sdz\BlogBundle\Bigbrother;
use Symfony\Component\EventDispatcher\Event;

class MessagePostEvent extends Event
{
}

```

C'est tout simplement une classe vide qui étend la classe Event du composant EventDispatcher.

Ensuite, il faut rajouter la spécificité de notre évènement. On a dit que le but de la fonctionnalité BigBrother est de censurer le message de certains utilisateurs, on a donc deux informations à transmettre du déclencheur au listener : le message et l'utilisateur qui veut le poster. On doit donc rajouter ces deux attributs à l'évènement :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Bigbrother/MessagePostEvent.php

namespace Sdz\BlogBundle\Bigbrother;
use Symfony\Component\EventDispatcher\Event;
use Symfony\Component\Security\Core\User\UserInterface;

class MessagePostEvent extends Event
{
    protected $message;
    protected $user;
    protected $autorise;

    public function __construct($message, UserInterface $user)
    {
        $this->message = $message;
        $this->user = $user;
    }

    // Le listener doit avoir accès au message
    public function getMessage()
    {
        return $this->message;
    }

    // Le listener doit pouvoir modifier le message
    public function setMessage($message)
    {
        return $this->message = $message;
    }

    // Le listener doit avoir accès à l'utilisateur
    public function getUser()
    {
        return $this->user;
    }

    // Pas de setUser, le listener ne peut pas modifier l'auteur du
    // message !
}

```

Faites attention aux getters et setters, vous devez les définir soigneusement en fonction de la logique de votre évènement :

- Un getter doit tout le temps être défini sur vos attributs. Car si votre listener n'a pas besoin d'un attribut (ce qui justifierait l'absence de getter), alors l'attribut ne sert à rien !
- Un setter ne doit être défini que si le listener peut modifier la valeur de l'attribut. Ici c'est le cas du message. Cependant, on interdit au listener de modifier l'auteur du message, cela n'aurait pas de sens.

Déclencher l'évènement

Déclencher et utiliser un évènement se fait assez naturellement lorsqu'on a bien défini l'évènement et ses attributs. Reprenons le code de l'action du contrôleur `BlogController` qui permet d'ajouter un article. Voici comment on l'adapterait pour déclencher l'évènement avant l'enregistrement effectif de l'article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;
use Sdz\BlogBundle\Bigbrother\BigbrotherEvents;
use Sdz\BlogBundle\Bigbrother\MessagePostEvent;
// ...

class BlogController extends Controller
{
    public function ajouterAction()
    {
        // ...

        if ($form->isValid()) {

            // On crée l'évènement avec ses 2 arguments
            $event = new MessagePostEvent($article->getContenu(),
                $article->getUser());

            // On déclenche l'évènement
            $this->get('event_dispatcher')
                ->dispatch(BigbrotherEvents::onMessagePost, $event);

            // On récupère ce qui a été modifié par le ou les listeners,
            // ici le message
            $article->setContenu($event->getMessage());

            // On continue l'enregistrement habituel
            $em = $this->getDoctrine()->getManager();
            $em->persist($article);
            $em->flush();

            // ...
        }
    }
}
```

C'est tout pour déclencher un évènement ! Vous n'avez plus qu'à reproduire ce comportement la prochaine fois que vous créerez une action qui permet aux utilisateurs d'ajouter un nouveau message (livre d'or, messagerie interne, etc.).

Écouter l'évènement

Comme vous avez pu le voir, on a déclenché l'évènement alors qu'il n'y a pas encore de listener. Cela ne pose pas de problème, bien au contraire : cela va nous permettre par la suite d'ajouter un ou plusieurs listeners qui seront alors exécutés au milieu de notre code. Ça, c'est du découplage !

Pour aller jusqu'au bout de l'exemple, voici ma proposition pour un listener. C'est juste un exemple, ne le prenez pas pour argent comptant :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Bigbrother/CensureListener.php

namespace Sdz\BlogBundle\Bigbrother;
use Symfony\Component\Security\Core\User\UserInterface;

class CensureListener
{
    // Liste des id des utilisateurs à surveiller
    protected $liste;
    protected $mailer;

    public function __construct(array $liste, \Swift_Mailer $mailer)
    {
        $this->liste = $liste;
        $this->mailer = $mailer;
    }

    // Méthode « reine » 1
    protected function sendEmail($message, UserInterface $user)
    {
        $message = \Swift_Message::newInstance()
            ->setSubject("Nouveau message d'un utilisateur surveillé")
            ->setFrom('admin@votresite.com')
            ->setTo('admin@votresite.com')
            ->setBody("L'utilisateur surveillé '". $user->getUsername() . "' a posté le message suivant : '" . $message . "'");
        $this->mailer->send($message);
    }

    // Méthode « reine » 2
    protected function censureMessage($message)
    {
        // Ici, totalement arbitraire :
        $message = str_replace(array('top secret', 'mot interdit'), '', $message);

        return $message;
    }

    // Méthode « technique » de liaison entre l'évènement et la fonctionnalité reine
    public function onMessagePost(MessagePostEvent $event)
    {
        // On active la surveillance si l'auteur du message est dans la liste
        if (in_array($event->getUser()->getId(), $this->liste)) {
            // On envoie un e-mail à l'administrateur
            $this->sendEmail($event->getMessage(), $event->getUser());

            // On censure le message
            $message = $this->censureMessage($event->getMessage());
            // On enregistre le message censuré dans l'event
            $event->setMessage($message);
        }
    }
}

```

Et bien sûr, la définition du service qui convient :

Code : YAML

```

# src/Sdz/BlogBundle/Resources/config/services.yml

services:

```

```

sdzblog.censure_listener:
    class: Sdz\BlogBundle\Bigbrother\CensureListener
    arguments: [[1, 2], @mailer]
    tags:
        - { name: kernel.event_listener, event: sdzblog.bigbrother.post_message, method: onMessagePost }

```

J'ai mis ici arbitrairement une liste [1, 2] pour les id des utilisateurs à surveiller, mais vous pouvez personnaliser cette liste ou même la rendre dynamique.

Allons un peu plus loin

Le gestionnaire d'évènements est assez simple à utiliser, et vous connaissez en réalité déjà tout ce qu'il faut savoir. Mais je ne pouvais pas vous laisser sans vous parler de trois points supplémentaires, qui peuvent être utiles.

Étudions donc les souscripteurs d'évènements, qui peuvent se mettre à écouter un évènement de façon dynamique, l'ordre d'exécution des listeners, ainsi que la propagation des évènements.

Les souscripteurs d'évènements

Les souscripteurs sont assez semblables aux listeners. La seule différence est la suivante : au lieu d'écouter toujours le même évènement défini dans un fichier de configuration, un souscripteur peut écouter dynamiquement un ou plusieurs évènements.

Concrètement, c'est l'objet souscripteur lui-même qui va dire au gestionnaire d'évènements les différents évènements qu'il veut écouter. Pour cela, un souscripteur doit implémenter l'interface `EventSubscriberInterface`, qui ne contient qu'une seule méthode : `getSubscribedEvents()`. Vous l'avez compris, cette méthode doit retourner les évènements que le souscripteur veut écouter.

Voici un simple exemple d'un souscripteur arbitraire dans notre bundle Blog :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Event/TestSubscriber.php

namespace Sdz\BlogBundle\Event;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class TestSubscriber implements EventSubscriberInterface
{
    // La méthode de l'interface que l'on doit implémenter, à définir
    // en static
    static public function getSubscribedEvents()
    {
        // On retourne un tableau « nom de l'évènement » => « méthode
        // à exécuter »
        return array(
            'kernel.response' => 'onKernelResponse',
            'store.order'      => 'onStoreOrder',
        );
    }

    public function onKernelResponse(FilterResponseEvent $event)
    {
        // ...
    }

    public function onStoreOrder(FilterOrderEvent $event)
    {
        // ...
    }
}

```

Bien sûr, il faut ensuite déclarer ce souscripteur au gestionnaire d'évènements. À ce jour, il n'est pas possible de le faire grâce à un tag dans une définition de service. Il faut donc le faire manuellement, comme on a appris à le faire avec un simple listener :

Code : PHP

```
<?php
// Depuis un contrôleur ou autre

use Sdz\BlogBundle\Event\TestSubscriber;

// On récupère le gestionnaire d'évènements
$dispatcher = $this->get('event_dispatcher');

// On instancie notre souscripteur
$subscriber = new TestSubscriber();

// Et on le déclare au gestionnaire d'évènements
$dispatcher->addSubscriber($subscriber);
```

L'ordre d'exécution des listeners

On peut définir l'ordre d'exécution des listeners grâce à un indice `priority`. Cet ordre aura ainsi une importance lorsqu'on verra comment stopper la propagation d'un évènement.

La priorité des listeners

Vous pouvez ajouter un indice de priorité à vos listeners, ce qui permet de personnaliser leur ordre d'exécution sur un même évènement. Plus cet indice de priorité est élevé, plus le listener sera exécuté tôt, c'est-à-dire avant les autres. Par défaut, si vous ne précisez pas la priorité, elle est de 0.

Vous pouvez la définir très simplement dans le tag de la définition du service. Ici, je l'ai définie à 2 :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdzblog.beta_listener:
        class: Sdz\BlogBundle\Beta\BetaListener
        arguments: ["2013-08-19"]
        tags:
            - { name: kernel.event_listener, event: kernel.response,
method: onKernelResponse, priority: 2 }
```

Et voici pour le cas où vous enregistrez votre listener en manipulant directement le gestionnaire d'évènements :

Code : PHP

```
<?php
$dispatcher->addListener('kernel.response', array($listener,
'onKernelResponse'), 2);
```

Vous pouvez également définir une priorité négative, ce qui aura pour effet d'exécuter votre listener relativement tard dans

l'évènement. Je dis bien relativement, car s'il existe un autre listener avec une priorité de -128 alors que le vôtre est à -64, alors c'est lui qui sera exécuté après le vôtre.

La propagation des évènements

Si vous avez l'œil bien ouvert, vous avez pu remarquer que tous les évènements qu'on a vus précédemment avaient deux méthodes en commun : `stopPropagation()` et `isPropagationStopped()`. Eh bien, vous ne devinerez jamais, mais la première méthode permet à un listener de stopper la propagation de l'évènement en cours !

La conséquence est donc directe : tous les autres listeners qui écoutaient l'évènement et qui ont une priorité plus faible *ne* seront *pas* exécutés. D'où l'importance de l'indice de priorité que nous venons juste de voir !

Pour visualiser ce comportement, je vous propose de modifier légèrement notre `BetaListener`. Rajoutez cette ligne à la fin de sa méthode `onKernelResponse` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Beta/BetaListener.php

// ...

public function onKernelResponse(FilterResponseEvent $event)
{
    // ...

    // On stoppe la propagation de l'évènement en cours (ici,
    kernel.response)
    $event->stopPropagation();
}
```

Actualisez une page. Vous voyez une différence ? La barre d'outils a disparu du bas de la page ! En effet, cette barre est ajoutée avec un listener sur l'évènement `kernel.response`, exactement comme notre mention « bêta ». Pour votre culture, il s'agit de `Symfony\Bundle\WebProfilerBundle\EventListener\WebDebugToolbarListener`.



La deuxième méthode, `isPropagationStopped()`, permet de tester si la propagation a été stoppée *dans le listener qui a stoppé l'évènement*. Les autres listeners n'étant pas exécutés du tout, il est évidemment inutile de tester la propagation dans leur code (à moins qu'eux-mêmes ne stoppent la propagation bien sûr).

En résumé

- Un évènement correspond à un moment clé dans l'exécution d'une page ou d'une action.
- On parle de *déclencher un évènement* lorsqu'on signale au gestionnaire d'évènements qu'un certain évènement vient de se produire.
- On dit qu'un listener *écoute un évènement* lorsqu'on signale au gestionnaire d'évènements qu'il faut exécuter ce listener dès qu'un certain évènement se produit.
- Un listener est une classe qui remplit une fonction, et qui écoute un ou plusieurs évènements pour savoir quand exécuter sa fonction.
- On définit les évènements à écouter via les tags du service listener.
- Il existe plusieurs évènements de base dans Symfony2, et il est possible de créer les nôtres.

Traduire son site

Maintenant que votre site est opérationnel, il faut penser à monter en puissance et conquérir le reste du monde ! Pour cela, il va falloir apprendre quelques langues supplémentaires à votre site, car malheureusement tout le monde ne parle pas français. Ce chapitre a donc pour objectif de mettre en place un site multilingue. Cela passera par :

- Traduire les messages qui apparaissent tels quels dans vos pages HTML ;
- Placer un peu de contenu dynamique au milieu d'un texte ;
- Afficher votre site dans différentes langues ;
- Traduire du contenu d'entités.

Quand vous avez créé votre site, vous avez mis certains textes directement dans vos templates Twig. Seulement, quand il est question de traduire le site, la question se pose : « Comment faire pour que ce texte, actuellement en dur dans le template, change selon la langue de l'utilisateur ? » Eh bien, nous allons rendre ce texte dynamique. Comment ? Nous avons tout un chapitre pour y répondre.

Vous êtes prêts ? Allons-y !

Introduction à la traduction

Le principe

Si demain on vous demande de traduire un document du français vers une langue étrangère, de quoi auriez-vous besoin ? En plus de votre courage, il vous faut impérativement ces trois informations :

- Ce qu'il faut traduire ;
- Dans quelle langue ;
- Ainsi qu'un dictionnaire si besoin.

Ensuite, la méthodologie est plutôt classique : on prend le texte original pour d'abord traduire les mots inconnus, puis on traduit les phrases en respectant la syntaxe de la langue cible.

En effet, quand nous commençons l'apprentissage d'une langue étrangère, nous cherchons le mot exact dans notre dictionnaire, sans imaginer qu'il s'agisse d'un adjectif accordé ou d'un verbe conjugué. On ne risque donc pas de trouver cette orthographe exacte. Symfony n'ayant pas d'intelligence artificielle, il va reproduire ce comportement systématiquement.

Ce qu'il en est avec Symfony2

Le service de traduction ne va donc pas s'embarrasser de considérations de syntaxe ou de grammaire ; c'est dû au fait qu'il ne s'agit pas d'un traducteur sémantique. Il n'analyse pas le contenu de la phrase — ni même ne regarde si ce qu'on lui fournit en est une. Il se chargera de traduire une chaîne de caractères d'une langue à l'autre, en la comparant avec un ensemble de possibilités. Notez bien cependant que la casse, tout comme les accents, sont importants. Symfony va vraiment chercher pour une correspondance *exacte* de la chaîne à traduire. C'est pourquoi on ne parle pas de **dictionnaire**, mais de **catalogue**.

Si d'un côté c'est plutôt rassurant car il ne peut pas faire d'erreur, l'autre côté implique évidemment que cette responsabilité *vous incombe*, car c'est vous qui allez écrire le catalogue pour Symfony ! Donc autant vous assurer que vous connaissez bien la langue dans laquelle vous allez traduire.

La langue source, c'est celle que vous avez déjà utilisée dans vos templates jusqu'à maintenant, donc probablement le français. Comme on l'a vu juste avant, Symfony n'allant chercher que la correspondance exacte, il n'y a pas réellement besoin de la spécifier.

Quant à la langue cible, elle est en général demandée par l'utilisateur, parfois sciemment (quand il clique sur un lien qui traduit la page sur laquelle il se trouve), parfois implicitement (quand il suit un lien depuis un moteur de recherche, lien qui est dans sa langue), et parfois à son insu (les navigateurs envoient, dans l'en-tête des requêtes, la (les) locale(s) préférée(s) de l'utilisateur ; la locale utilisée sur un site est souvent stockée dans la session, liée à un cookie, qui voyage donc aussi à chaque requête).



On parle de **locale** pour désigner non seulement la langue de l'utilisateur, mais aussi d'autres paramètres régionaux, comme le format d'affichage de sommes d'argent (et donc la devise), de dates, etc. La locale contient un code de langue ainsi qu'une éventuelle indication du pays. Exemples : `fr` pour le français, `fr_CH` pour le français de Suisse, `zh_Hant_TW` pour le chinois traditionnel de Taïwan.

Les locales sont composées de codes de langue, au [format ISO639-1](#), puis éventuellement d'un sous-tiret (`_`) et du code du pays au [format ISO3166-2 Alpha-2](#).



Et s'il n'y a pas de traduction dans la locale demandée ?

Symfony possède un mécanisme qui permet d'afficher quelque chose par défaut. Imaginons que vous arriviez sur un site principalement anglophone géré par des Québécois, et ceux-ci, par égard aux Français, sont en train de préparer une version spéciale en « français de France ». Cependant, tout le site n'est pas encore traduit.

Vous demandez la traduction pour votre locale `fr_FR` du texte « `site.devise` » :

1. Ce qui est déjà prévu pour la locale `fr_FR` vous sera retourné ;
2. Ce qui n'est pas encore « traduit », mais existe en « français général » (locale `fr`) : c'est cette version qui sera envoyée ;
3. Ce qui n'est pas du tout traduit en français, mais l'est en anglais, est affiché en anglais ;
4. Ce qui ne possède aucune traduction est affiché tel quel, ici « `site.devise` ». Dans ce cas, quand c'est le texte original qui est affiché, c'est que vous avez oublié la traduction de ce terme.

Ainsi, il n'y aura jamais de vide là où vous avez spécifié du texte à traduire.

Prérequis

Avant de partir dans la traduction de son site, il faut vérifier que Symfony travaillera correctement avec les langues, et notamment celle qui est utilisée par défaut. Comme nous sommes sur un site francophone, je vais partir du principe que la langue par défaut de votre site est le français, et la locale `fr`.

Configuration

Pour savoir quelle est la langue sur le site (au cas où il ne serait pas possible d'afficher celle que le client souhaite), Symfony utilise un paramètre appelé `locale`, comme nous l'avons vu plus haut. La locale pour le français est `fr`, en minuscules, et il nous faut la définir comme locale par défaut dans le fichier `app/config/parameters.yml`. Ouvrez donc ce fichier et effectuez-y la manipulation mentionnée.

Code : YAML

```
# app/config/parameters.yml

parameters:
    locale: fr # Mettez « fr » ici
```



Si ce paramètre n'est pas renseigné, le framework considérera qu'il travaille en anglais.

On va ensuite utiliser ce paramètre `locale` dans la configuration :

Code : YAML - (extrait)

```
# app/config/config.yml

framework:
    # On définit la langue par défaut pour le service de traduction
    # Décommenter la ligne, et vérifier qu'elle est bien ainsi
    translator: { fallback: %locale% }

    # ...

    # Vérifier cette ligne aussi, pour la langue par défaut de
    # l'utilisateur
    # C'est celle qui sera utilisée si l'internaute ne demande rien
    default_locale: %locale%
```

Votre application sait maintenant que vous travaillez sur un site qui, à la base, est en français.

Mise en place d'une page de test

Pour la suite du chapitre, nous avons besoin d'une page sur laquelle réaliser nos tests. Je vous invite donc à créer la même que moi, afin qu'on s'y retrouve.

Tout d'abord voici la route, à rajouter au fichier de routes de l'application. On va la mettre dans le `routing_dev.yml`, car d'une part c'est une route de test (pas destinée à nos futurs visiteurs !), et d'autre part le fichier de routes de notre bundle est préfixé par `/blog` qu'on ne veut pas forcément ici. Voici donc la route en question :

Code : YAML

```
# app/config/routing_dev.yml

SdzBlogBundle_traduction:
    pattern: /traduction/{name}
    defaults: { _controller: SdzBlogBundle:Blog:traduction }
```

Pour que la route fonctionne, il nous faut aussi créer l'action qui lui correspond, dans le contrôleur Blog. Une fois de plus, je vous mets le code, rien de bien sorcier :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function traductionAction($name)
{
    return $this->render('SdzBlogBundle:Blog:traduction.html.twig',
array(
    'name' => $name
));
}
```

Et comme indiqué dans le contrôleur, il nous faut la vue `traduction.html.twig`, la voici :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/traduction.html.twig #}

<html>
    <body>
        Hello {{ name }} !
    </body>
</html>
```

C'est bon, on va pouvoir mettre la main à la pâte !

Bonjour le monde

Actuellement, quand vous accédez à `/traduction/winzou`, la page qui s'affiche ne contient que « Hello winzou! », et ce quelle que soit la langue. Nous allons faire en sorte qu'en français nous ayons « Bonjour winzou! », c'est-à-dire que le « Hello » soit traduit en « Bonjour ».

Dire à Symfony « Traduis-moi cela »

La traduction est possible dans Symfony à deux endroits : dans les contrôleurs et dans la vue. Cette dernière option est la plus conseillée, car c'est dans les vues que se situe l'affichage et donc bien souvent le texte à traduire.

Le filtre Twig {{ ' ' | trans }}

Un filtre est, dans le langage Twig, une fonction destinée à formater/modifier une valeur. C'est donc tout à fait adapté à la traduction de texte, car modifier le texte pour qu'il soit dans une autre langue est une transformation comme une autre !

Plus précisément dans notre cas, c'est le filtre `trans` que l'on va utiliser. La syntaxe est la suivante : `{{ 'ma chaîne'|trans }}` ou encore `{{ ma_variable|trans }}`. Ce filtre est prévu pour s'appliquer sur des variables ou des courtes chaînes, voici un exemple dans un contexte :

Code : HTML & Django

```
<div>
  <p>{{ message|trans }}</p>
  <button>{{ 'cancel'|trans }}</button>
  <button>{{ 'validate'|trans }}</button>
</div>
```

La balise de bloc Twig { % trans % }

Une autre possibilité de traduction depuis la vue consiste à encadrer tous les textes dans des blocs `{ % trans %} ... { % endtrans %}`. Ce bloc permet de traduire du texte brut, mais attention il est impossible d'y mettre autre chose que du texte. Balises HTML, code Twig, etc. sont interdits ici. Une des utilisations les plus parlantes est pour les conditions générales d'utilisation d'un site, où il y a de gros paragraphes avec du texte brut, voici un exemple :

Code : HTML & Django

```
<p>
  { % trans %}Lorem ipsum dolor sit amet, consectetur adipiscing
  elit. Curabitur
    quam nisi, sollicitudin ut rhoncus semper, viverra in augue.
  Suspendisse
    potenti. Fusce sit amet eros tortor. Class aptent taciti sociosqu
  ad litora
    torquent per conubia nostra, per inceptos himenaeos. Ut arcu
  justo, tempus sit
    amet condimentum vel, rhoncus et ipsum. Mauris nec dui nec purus
  euismod
    imperdiet. Cum sociis natoque penatibus et magnis dis parturient
  montes,
    nascetur ridiculus mus. Mauris ultricies euismod dolor, at
  hendrerit nulla
    placerat et. Aenean tincidunt enim quam. Aliquam cursus lobortis
  odio, et
    commodo diam pulvinar ut. Nunc a odio lorem, in euismod eros.
  Donec viverra
    rutrum ipsum quis consectetur. Etiam cursus aliquam sem eget
  gravida. Sed id
    metus nulla. Cras sit amet magna quam, sed consectetur odio.
  Vestibulum feugiat
    justo at orci luctus cursus.{ % endtrans %}
</p>
<p>
  { % trans %}Vestibulum sollicitudin euismod tellus sed rhoncus.
  Pellentesque
    habitant morbi tristique senectus et netus et malesuada fames ac
  turpis
    egestas. Duis mattis feugiat varius. Aenean sed rutrum purus. Nam
  eget libero
    lorem, ut varius purus. Etiam nec nulla vitae lacus varius
  fermentum. Mauris
    hendrerit, enim nec posuere tempus, diam nisi porttitor lacus, at
```

```
placerat
    elit nulla in urna. In id nisi sapien. { % endtrans % }
</p>
```

D'accord, l'exemple n'est pas vraiment bon, mais cela illustre l'utilisation. On a de gros pavés de texte, et je vous laisse regarder et réfléchir à ce que cela aurait représenté avec la solution précédente du filtre. 

Le service translator

Parfois, vous devrez malgré tout réaliser quelques traductions depuis le contrôleur, dans le cas d'inscriptions dans un fichier de log, par exemple. Dans ce cas il faut faire appel au service `translator`, qui est le service de traduction que la balise et le filtre Twig utilisent en réalité. Son utilisation directe est très aisée, voyez par vous-mêmes :

Code : PHP

```
<?php
// Depuis un contrôleur

// On récupère le service translator
$translator = $this->get('translator');

// Pour traduire dans la locale de l'utilisateur :
$texteTraduit = $translator->trans('Mon message à inscrire dans les
logs');
```

Notre vue



Bon, et on fait comment, finalement, pour traduire notre « Hello » en « Bonjour » ?

C'est vrai, revenons-en à nos moutons. Adaptons donc le code de notre vue en rajoutant le filtre `trans` de Twig pour qu'il traduise notre « Hello » :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/traduction.html.twig #-}

<html>
    <body>
        {{ 'Hello'|trans }} {{ name }} !
    </body>
</html>
```

Accédez à nouveau à `/traduction/winzou` via l'environnement de développement.



Eh, mais... c'est encore « Hello » qui s'affiche ! Pourtant on s'est bien mis en français, non ?

C'est exact ! Mais rappelez-vous la méthode pour faire une traduction. Il faut savoir quoi traduire, ça c'est OK, il faut savoir dans quelle langue le traduire, ça c'est OK, la locale de l'utilisateur est automatiquement définie par Symfony2. Il nous manque donc... le dictionnaire !

En effet, on n'a pas encore renseigné Symfony sur comment dire « Hello » en français. Les fichiers qui vont le lui dire s'appellent des catalogues, nous y venons.

Le catalogue

Vous l'aurez compris, le catalogue est l'endroit où l'on va associer la chaîne à traduire avec sa version en langue étrangère. Si vous avez créé votre bundle grâce à la commande `generate:bundle`, alors Symfony2 vous a créé automatiquement un catalogue exemple, c'est le fichier enregistré dans le dossier `Resources/translations` du bundle, et qui contiendra par la suite les paires de type 'Ma chaîne en français' <=> 'My string in English'.

Les formats de catalogue

 Les exemples sont pour traduire de l'anglais au français, *et non l'inverse*.

Le format XLIFF

Symfony recommande le XLIFF, une application du XML. C'est pourquoi, dans les bundles générés avec la ligne de commande, vous trouvez ce fichier `Resources/translations/messages.fr.xlf` qui contient l'exemple suivant que je commente :

Code : XML

```
<!-- src/Sdz/BlogBundle/Resources/translations/messages.fr.xlf -->

<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext"
original="file.ext">
        <body>
            <trans-unit id="1">
                <!-- La chaîne source, à traduire. C'est celle que l'on
utilisera :
                {%
                    trans %}Symfony2 is great{%
                    endtrans %
                ou
                {{ 'Symfony2 is great'|trans }}

                ou
                $translator->trans('Symfony2 is great')) -->
                <source>Symfony2 is great</source>

                <!-- La chaîne cible, traduction de la chaîne ci-dessus -->
                <target>J'aime Symfony2</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

L'avantage du XLIFF est qu'il s'agit d'un format officiel, dont vous pouvez faire valider la structure en plus de la syntaxe. De plus, du fait du support natif de XML par PHP, il est facile de le modifier via PHP, donc de créer une interface dans votre site pour modifier les traductions. En revanche, le désavantage du XLIFF est celui du XML de façon générale : c'est très verbeux, c'est-à-dire que pour traduire une seule ligne il nous en faut vingt.

Le format YAML

Voici l'équivalent YAML du fichier `messages.fr.xlf` précédent :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/messages.fr.yml
```

```
# La syntaxe est : « la chaîne source: la chaîne cible »  
Symfony2 is great: J'aime Symfony2
```

Vous allez me dire que vu sa simplicité on se demande pourquoi le XLIFF existe. Je suis d'accord avec vous, pour l'utilisation simple que nous allons faire, le YAML est meilleur, car on distingue bien mieux les chaînes à traduire. Si c'est votre choix également, pensez à supprimer le fichier `messages.fr.xls` afin qu'il n'interfère pas. Enfin, gardez en tête le format XLIFF si vous souhaitez manipuler vos traductions en PHP, car c'est plus délicat en YAML.

Attention également lorsque vous avez des deux-points (« : ») dans votre chaîne source, vu la syntaxe utilisée dans le fichier YAML vous vous doutez qu'il ne va pas apprécier. Il faut dans ce cas encadrer la chaîne source par des guillemets, comme ceci :

Code : YAML

```
"Phone number": Numéro de téléphone :
```

Notez que vous pouvez ne pas encadrer la chaîne cible avec les guillemets, bien que celle-ci contienne également les deux-points. Cela est dû à la syntaxe du YAML lui-même : la fin de la chaîne cible est le retour à la ligne, donc impossible à confondre avec les deux-points. 😊

Le format PHP

Moins utilisé, ce format est néanmoins possible. La syntaxe est celle d'un simple tableau PHP, comme ceci :

Code : PHP

```
<?php  
// src/Sdz/BlogBundle/Resources/translations/messages.fr.php  
  
return array(  
    'Symfony2 is great' => 'J\'aime Symfony2',  
);
```

La mise en cache du catalogue

Quelque soit le format que vous choisissez pour vos catalogues, ceux-ci seront mis en cache pour une utilisation plus rapide dans l'environnement de production. En effet, ne l'oubliez pas, si Symfony2 a la gentillesse de régénérer le cache du catalogue à chaque exécution dans l'environnement de développement, il ne le fait pas en production. Cela signifie que vous devez vider manuellement le cache pour que vos changements s'appliquent en production. Pensez-y avant de chercher des heures pourquoi votre modification n'apparaît pas.

Vous pouvez choisir le format qui vous convient le mieux, mais dans la suite du cours je vais continuer avec le format YAML, qui possède quelques autres avantages intéressants dont je parle plus loin.

Notre traduction

Maintenant qu'on a vu comment s'organisait le catalogue, on va l'adapter à nos besoins. Créez le fichier `messages.fr.yml` nécessaire pour traduire notre « Hello » en français. Vous devriez arriver au résultat suivant sans trop de problèmes :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/messages.fr.yml
```

```
# On veut traduire « Hello » en « Bonjour »
Hello: Bonjour
```

Pour tester, videz votre cache et rechargez la page [/traduction/winzou](#). Ça marche ! Vous lisez désormais « Bonjour winzou ! ». Et si vous changez le paramètre dans `app/config/parameters.yml` pour que la locale par défaut soit à nouveau l'anglais, vous avez à nouveau « Hello winzou ! ».

 Chaque fois que vous créez un nouveau fichier de traduction, il vous faut rafraîchir votre cache avec la commande `cache:clear`, que vous soyez dans l'environnement de production ou de développement. En effet, si le mode `dev` permet de prendre en compte les modifications du catalogue sans vider le cache, ce n'est pas le cas pour la création d'un fichier de catalogue !

 Si vous avez une erreur du type « `Input is not proper UTF-8, indicate encoding !` », n'oubliez pas de bien encoder tous vos fichiers en UTF-8 sans BOM.

Ajouter un nouveau message à traduire

On se doute bien que l'on n'aura pas qu'une seule chaîne à traduire sur tout un site. Il va falloir que tout ce qu'on souhaite voir s'afficher dans la langue de l'utilisateur soit renseigné dans le catalogue. Pour chaque nouvelle chaîne, on ajoute une unité de traduction :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/messages.fr.yml
Hello: Bonjour
Bye: Au revoir
```

Ainsi, à chaque fois que vous avez une nouvelle traduction à ajouter, vous ajoutez une ligne.

Extraire les chaînes sources d'un site existant

 Mais moi j'ai déjà un site tout prêt en français, je dois trouver toutes les chaînes sources et les copier à la main ? 😞

Justement, non. Vous allez devoir ajouter les balises et/ou filtres Twig de traduction dans vos vues, ça, c'est inévitable. Mais une fois que ce sera fait, les concepteurs de Symfony ont pensé aux personnes dans votre cas, et ont développé un outil permettant d'extraire toutes les chaînes entre balises `{% trans %}` et celles avec le filtre `|trans`.

Cet outil se présente sous forme d'une commande, il s'agit de `translation:update`. Sa version complète est la suivante : `translation:update [--prefix="..."] [--output-format="..."] [--dump-messages] [-force] locale bundle`. Cette commande va lire toutes les vues du bundle spécifié, et compilera toutes les chaînes sources dans un catalogue. Vous n'aurez plus qu'à définir les chaînes cibles.

 Si cela paraît être la commande miracle, elle ne fonctionne cependant pas pour extraire les chaînes sources des contrôleurs, donc utilisées avec `<?php $this->get('translator')->trans(/* ... */)`, ni sur le contenu des variables traduites avec `{ { maVariable|trans } }` (car il ne connaît pas la valeur de la variable `maVariable`!). Et bien entendu, n'oubliez pas de balise/filtre Twig de traduction !

La commande `translation:update` est du même genre que `doctrine:schema:update`, dans le sens où il vous faut choisir de donner (en plus des deux paramètres obligatoires) soit `--dump-messages`, soit `--force` pour qu'elle

fasse réellement quelque chose. Cela permet de vérifier le résultat avant qu'elle n'écrive effectivement dans le catalogue.

- La première option `--dump-messages` affiche tous les messages dans la console, plus ou moins tels qu'ils seront dans un catalogue en YAML (c'est pour cela que c'est le format de sortie par défaut). Elle tient compte des messages déjà traduits, donc pas de souci que votre précédent travail soit écrasé. Cela vous permet aussi de passer d'un format à l'autre si vous aviez commencé le travail en réutilisant le fichier `messages.fr.xls` du bundle par exemple.
- La seconde option `--force` effectue la mise à jour des catalogues, tout en conservant une sauvegarde des versions précédentes.

Par défaut, les extractions sont de type `chaîne source`: `__ chaîne source`. En effet, Symfony ne peut deviner comment traduire la chaîne source, il la remet donc comme chaîne cible, mais en la préfixant avec `__`. Avec l'option `--prefix="..."`, vous pouvez changer la partie `__` par ce que vous voulez.

Il est temps de passer à la pratique, exécutons cette commande pour extraire les messages de notre bundle, et regardez ce qu'il en sort :

Code : Console

```
C:\wamp\www\Symfony> php app/console translation:update --dump-messages --force fr SdzBlogBundle
Generating "fr" translation files for "SdzBlogBundle"
Parsing templates
Loading translation files

Displaying messages for domain messages:

Hello: Bonjour
Bye: 'Au revoir'
'Symfony2 is great': 'J''aime Symfony2'

Writing files
```

Comme je vous l'avais mentionné, les chaînes cibles sont identiques aux chaînes sources mais avec un préfixe. À l'aide de votre éditeur préféré, cherchez les occurrences de `__` (ou du préfixe que vous avez défini vous-mêmes) dans vos catalogues pour les mettre en surbrillance. Vous ciblerez ainsi très rapidement les nouveautés, c'est-à-dire ce que vous devez traduire !

Lorsque la commande affiche `Displaying messages for domain messages`, elle se contente d'afficher tous les messages du domaine, après son travail d'extraction. Les « Hello » et « Bye » viennent de notre fichier YAML, et l'autre ligne du fichier XLIFF déjà existant. D'ailleurs, le générateur a remis cette valeur dans le fichier YAML (format par défaut), vous pouvez donc supprimer le fichier XLIFF sereinement (si vous avez choisi le YAML bien entendu). Vous noterez d'ailleurs que cette valeur a été échappée, même si ici ce n'est pas utile, vous pouvez supprimer toutes les `quotes` sans problèmes.

Enfin, vous noterez que les chaînes cibles ne sont pas préfixées, car nous les avons déjà traduites !



Symfony s'occupe automatiquement des éventuels doublons, vous ne devriez normalement plus en avoir après avoir utilisé la commande `translations:update` avec l'option `--force`.

Traduire dans une nouvelle langue

Pour conquérir le monde, ce qui reste notre but à tous, c'est bien de parler anglais, mais il ne faut pas s'arrêter là ! On souhaite maintenant traduire les messages également en allemand. Il faut alors tout simplement créer le catalogue adéquat, mais on peut se simplifier la vie : dupliquez le fichier `messages.fr.yml` et nommez la copie `messages.de.yml`. Ensuite, vous n'avez plus qu'à y modifier les chaînes cibles :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/messages.de.yml
Hello: Guten Tag
```

Vous pouvez dès à présent tester le bon fonctionnement de l'allemand en changeant le paramètre dans `app/config/parameters.yml`, après avoir vidé votre cache bien entendu (nous avons ajouté un fichier dans le catalogue).

 Je vous conseille vivement de bien préparer — si ce n'est de terminer — les catalogues pour une locale, puis de les dupliquer pour les autres, et non de travailler sur toutes les locales à la fois. Vous éviterez ainsi les casse-têtes de traductions incomplètes (manque de chaînes sources surtout). La traduction est un processus qui doit se faire tout à la fin de la réalisation d'un site.

Récupérer la locale de l'utilisateur

Déterminer la locale

Jusqu'à présent, pour changer la locale de notre site nous avons modifié à la main le fichier de configuration. Bien entendu, ce n'est pas une solution viable, vous n'allez pas rester derrière votre PC à changer la locale dès qu'un internaute étranger arrive sur votre site !

La question se pose donc de savoir comment adapter la locale à l'utilisateur qui navigue sur votre site. Souvenez-vous, j'ai précisé un peu plus haut qu'il y avait plusieurs moyens de connaître la locale de l'utilisateur. Je vous les rappelle ici :

1. L'utilisateur clique sur un lien qui traduit la page sur laquelle il se trouve ;
2. L'utilisateur envoie ses préférences dans les en-têtes des requêtes ;
3. Les paramètres par défaut.

L'ordre correspond à la priorité observée par Symfony.

La plupart des sites multilingues affichent la locale dans l'URL (exemple : `http://www.site.com/fr`). La locale ne dépend donc pas de l'utilisateur, mais de l'adresse à laquelle il se trouve.

Routing et locale

Vous l'avez peut-être déjà compris, pour que la locale apparaisse dans les URL, il va falloir ajouter un paramètre dans nos URL.

Le paramètre d'URL

Nous l'avons vu dans le chapitre sur le routeur : certains paramètres de route bénéficient d'un traitement spécial de la part de Symfony2. Et devinez quoi ? Il y en a un prévu pour récupérer la locale ! Il s'agit du paramètre `_locale`, qui a déjà été mentionné dans le chapitre sur le routeur.



Mais les paramètres, on doit les traiter dans les contrôleurs, normalement, non ? Donc on va devoir modifier toutes nos actions, en plus des routes ?

Justement non, c'est en cela que le paramètre `_locale` (et d'autres) sont spéciaux. En effet, Symfony sait quoi faire avec ces paramètres, vous n'avez donc pas à les récupérer dans vos actions — sauf si le traitement diffère sensiblement en fonction de ce paramètre — ni le mettre vous-mêmes en session, ni quoi que ce soit.

Voici ce que cela donne sur notre route de test :

Code : YAML

```
# app/config/routing_dev.yml

SdzBlogBundle_traduction:
    pattern: '/{_locale}/traduction/{name}'
    defaults: { _controller: SdzBlogBundle:Blog:traduction }
```



Les paramètres de route sont différents des variables que vous pouvez passer en GET. Ainsi, l'URL `/traduction/winzou?_locale=ur` ne traduira pas votre page (en ourdou dans le cas présent), car ici `_locale` n'est pas un paramètre de route.

Vous pouvez déjà tester en essayant `/fr/traduction/winzou` ou `/de/traduction/winzou`, le contenu de la page est bien traduit.

Attention par contre, si vous allez sur `/en/traduction/winzou`, vous avez toujours « Bonjour ». En effet, pour l'instant on n'a pas de catalogue pour l'anglais. Alors effectivement le « Hello » inscrit dans la vue est déjà en anglais, mais ça, Symfony ne le sait pas ! Car on aurait pu mettre n'importe quoi à la place de « Hello », comme on le verra un peu plus loin. Bref, comme il n'y a pas de catalogue correspondant à la langue demandée, Symfony affiche la page dans la langue *fallback* (langue de repli en français), définie dans le fichier de configuration `config.yml`, qui est dans notre cas le français. 😐



Mais cela veut dire qu'on va éditer *tous* nos fichiers de routing et prendre chaque route *une à une* ?

Non, rassurez-vous ! Il y a au moins un moyen plus rapide de faire cela, qu'on a aussi vu dans le chapitre sur les routes... il s'agit de l'utilisation d'un préfixe ! Voyez par vous-mêmes comment rajouter le paramètre `_locale` sur tout notre blog :

Code : YAML

```
# app/config/routing.yml

SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix:  /{_locale}/blog # Ici, on ajoute {_locale} au
    préfixe !
```



Vous pouvez désormais demander vos pages de blog en différentes langues selon l'URL : `/fr/blog`, `/en/blog`, etc. Bien entendu, pour que ce soit parfaitement opérationnel, vous devez généraliser l'utilisation du filtre ou de la balise Twig `trans`, et traduire les textes dans les catalogues correspondants.

Il y a actuellement un problème avec les routes de FOSUserBundle quand on utilise cette solution de préfixer depuis `app/config/routing.yml`. La route `fos_user_security`, quand elle est préfixée, n'est plus liée à l'action. Ainsi, pour les routes de FOSUserBundle, il vaut mieux dupliquer le fichier de routing du bundle dans votre `UserBundle` et y préfixer les routes qui en ont besoin. Vous n'avez plus qu'à importer le routing de votre `UserBundle` à la place de celui de FOSUserBundle, et vous n'avez pas besoin de mettre la partie du préfixe `{_locale}` à l'importation.

Il manque cependant un garde-fou à notre solution : avec une locale ainsi apparente, un petit malin peut très bien changer à la main la locale dans une URL, et arriver sur un site en traduction que vous ne pensiez pas être accessible. Veillez donc à limiter les locales disponibles en ajoutant les *requirements* pour ce paramètre. De fait, le routing final devrait ressembler à cela :

Code : YAML

```
# app/config/routing.yml

SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix:  /{_locale}/blog
    requirements:
        locale: en|fr # les locales disponibles, séparées par des
        pipes « | »
```



Soyons clairs : si vous avez des routes qui contiennent la locale, vous n'avez rien d'autre à faire. Ni manipuler la session, ni l'objet `request`, ni quoi que ce soit d'autre. Symfony s'en charge.

Les paramètres par défaut

Ces paramètres sont prévus pour éviter que l'internaute ne trouve ni aucun contenu, ni un contenu incompréhensible. Mais ils sont aussi définis pour que Symfony puisse fonctionner. Nous avons vu l'ordre de priorité dans les possibilités de passer la locale au framework. En fait, Symfony ne se base que sur la session, mais la remplit selon cet ordre de priorité. Seulement, il y a toujours un moment où l'internaute arrive pour la toute première fois sur notre site (ou une nouvelle fois après avoir entièrement nettoyé son cache navigateur, ce qui, pour le serveur, revient au même). Du coup, il faut bien des paramètres par défaut.

Au début de ce chapitre, nous avons vérifié et changé quelques paramètres dans `app/config/config.yml`. Reprenons le code de ce fichier un peu plus en détail, afin de mieux comprendre à quoi il sert :

Code : YAML

```
# app/config/config.yml

framework:
    # On définit la langue par défaut pour le service de traduction
    # Ce qui n'est pas disponible dans la locale de l'utilisateur
    # sera affiché dans celle spécifiée ici
    translator:      { fallback: %locale% }

    # ...

    # On initialise la locale de requête, celle par défaut pour
    # l'internaute arrivant pour la toute première fois sur votre
    # site
    default_locale: %locale%
```

Organiser vos catalogues

Quand il y a beaucoup de traductions, les fichiers deviennent vite difficiles à manipuler. Il faut parcourir beaucoup de lignes pour retrouver là où l'on souhaite faire une modification, et c'est contraire au mode de vie des informaticiens qui veut qu'on puisse se retrouver rapidement dans du code ou dans des fichiers. Je vais donc vous proposer des solutions pour alléger vos catalogues.

Utiliser des mots-clés plutôt que du texte comme chaînes sources

Voilà une solution intéressante, à vous de choisir de l'adopter pour toutes vos traductions ou juste les chaînes très longues. L'idée est d'utiliser, au lieu du texte dans la langue source, des mots-clés.

Plutôt qu'un long discours, je vous propose un petit exemple. Prenons une page statique avec pas mal de texte, ce qui implique beaucoup de texte dans le catalogue, par exemple :

Code : YAML

```
# Dans un catalogue

Le site où on apprend tout... à partir de zéro !: The website where
you learn it all... from scratch!
```

L'entrée du catalogue est composée de deux longues phrases, ce n'est pas terrible. Et je ne vous parle pas de son utilisation dans les vues :

Code : HTML & Django

```
{# Dans une vue #}
```

```
{% trans %}Le site où on apprend tout... à partir de zéro !{% trans %}  
{# ou #}  
{ { 'Le site où on apprend tout... à partir de zéro !'|trans }}
```

Passons maintenant à l'utilisation d'un mot-clé, vous allez tout de suite comprendre. Voici d'abord le catalogue :

Code : YAML

```
# Dans un catalogue  
  
site.devise: The website where you learn it all... from scratch!
```

Vous voyez déjà à quel point c'est plus léger !



Bien entendu, si le catalogue est léger, il n'y a rien de magique : vous devez dans ce cas utiliser deux catalogues. L'un pour l'anglais et l'autre pour le français !

Mais l'avantage se situe surtout dans les vues, où un mot-clé est plus synthétique qu'une longue phrase, utile pour ne pas se perdre au milieu du code HTML de votre vue. Voyez vous-mêmes :

Code : HTML & Django

```
{# Dans une vue #}  
  
{% trans %}site.devise{% trans %}  
{# ou #}  
{ { 'site.devise'|trans }}
```

Vous voyez : quelques mots-clés bien choisis pour résumer une phrase, séparés par des points, et vous avez déjà gagné en clarté dans vos vues et catalogues ! Cela est utilisable avec n'importe quel format de catalogue. N'hésitez pas à vous en servir copieusement.

Un des avantages également est de voir très rapidement une chaîne non traduite : au lieu du joli texte en français, vous aurez un « xxx.yyy » au milieu de votre page. Cela saute mieux aux yeux, et évite les oubliés !

Enfin, un mot sur la création de deux catalogues au lieu d'un seul. C'est en réalité plutôt une bonne chose, car cela permet non seulement de séparer le texte de tout le code HTML, mais cela permet aussi de mutualiser ! En effet, si vous servez d'un mot ou d'une phrase de façon récurrente sur votre site (la devise par exemple), celui-ci ne sera stocké qu'à un seul endroit, dans votre catalogue. Vous pourrez alors le modifier à un unique endroit, et les répercussions s'appliqueront partout sur votre site.

Nicher les traductions

C'est une possibilité qui découle de l'utilisation des mots-clés.



Cette possibilité n'est disponible que dans les catalogues au format YAML.

Si vous optez pour l'utilisation des mots-clés, ce que je vous conseille, vous arriverez très certainement à un résultat de ce genre :

Code : YAML

```
# Dans un catalogue
```

```
article.edit.title: Édition d'un article
article.edit.submit_button: Valider
article.show.edit_button: Éditer l'article
article.show.create_button: Créer un nouvel article
```

Ce qui était très clair avec une seule ligne, le devient déjà moins lorsqu'il y en a quatre, alors imaginez avec plus !

En bons développeurs avisés, vous avez tout de suite repéré les redondances, et votre envie de les factoriser est grande. Sachez que vous n'êtes pas seuls, les développeurs du YAML ont pensé à tout, voici comment optimiser votre catalogue :

Code : YAML

```
# Dans un catalogue

article:
    edit:
        title: Édition d'un article
        submit_button: Valider
    show:
        edit_button: Éditer l'article
        create_button: Créer un nouvel article
```

Quand Symfony va lire cette portion de YAML, il va remplacer chaque séquence « deux points – retour à la ligne – indentation » par un simple point, devenant ainsi l'équivalent de ce que vous aviez précédemment. Très pratique !

Côté utilisation, dans les vues ou avec le service `translator`, rien ne change. Vous utilisez toujours `{ { 'article.edit.title' | trans } }` par exemple.



Sachez que c'est une fonctionnalité du YAML, et non du service de traduction de Symfony2. Vous pouvez voir cette utilisation dans votre fichier de configuration `app/config/config.yml` par exemple !

Pour en revenir à l'organisation du catalogue avec ces mots-clés, je vous propose de toujours respecter une structure de ce genre :

Code : YAML

```
sdz: # Le namespace racine que vous utilisez
      blog: # Le nom du bundle, sans la partie Bundle
            article: # Le nom de l'entité ou de la section
                  list: # Les différents messages, pages et/ou actions
                  new: # Etc.
```

Permettre le retour à la ligne au milieu des chaînes cibles

Certains éditeurs ne gèrent pas le retour à la ligne automatique, et du coup, ce ne sont pas les chaînes sources trop longues qui posent problème, mais les chaînes cibles. Le parseur YAML fourni avec Symfony supporte une syntaxe intéressante qui permet d'éviter d'avoir à faire défiler horizontalement le contenu des catalogues.

Difficile d'expliquer cela sans un exemple, prenons la charte du Site du Zéro :

Code : YAML

```
# Dans un catalogue

charte:
```

```

titre: Mentions légales
donnee:
    # Le chevron « > » en début de chaîne indique que la chaîne
    # cible est sur
    # plusieurs lignes, mais les retours à la ligne ne seront
    # pas présents
    # dans le code HTML, car ils seront remplacés par des
    # espaces.
    # L'indentation doit être faite sur tout le paragraphe.
debut: >
    Le Site du Zéro recueille des informations (login, e-
mail) lors de
        votre enregistrement en tant que membre du site. Lors de
        votre
        connexion au site, un fichier "log" stocke les actions
        effectuées
            par votre ordinateur (via son adresse IP) au serveur.

    # La pipe « | » permet la même chose, mais les retours à la
    # ligne seront
    # présents dans le code HTML, et non remplacés par des
    # espaces.
    # Vous pouvez utiliser nl2br() sur une telle chaîne, cela
    # permet
    # d'avoir le code comme présenté ci-dessous (l'indentation
    # en moins).
fin: |
    Lorsque que vous vous connectez en tant que membre du
    Site du Zéro et
        que vous cochez la case correspondante, un cookie est
        envoyé à votre
            ordinateur afin qu'il se souvienne de votre login et de
        votre mot de
            passe. Ceci vous est proposé uniquement afin
        d'automatiser la
            procédure de connexion, et n'est en aucun cas utilisé
        par Simple IT à
            d'autres fins.

```

Avec la pipe et le chevron, vous pouvez donc faire tenir votre catalogue sur 80 caractères de large, ou tout autre nombre qui vous convient.

Utiliser des listes

Encore une possibilité du langage YAML qui peut s'avérer pratique dans le cas de catalogues !

Reprends l'exemple précédent de la charte pour en faire une liste. En effet, on rencontre souvent une série de paragraphes, dont certains seront supprimés, d'autres ajoutés, et il faut pouvoir le faire assez rapidement. Si vous n'utilisez pas de liste, et que vous supprimez la partie 2 sur 3, ou que vous ajoutez un nouveau paragraphe entre deux autres... vous devez soit adapter votre vue, soit renommer les parties et paragraphes. Bref, ce n'est clairement pas idéal.

Heureusement, il y a un moyen d'éviter cela en YAML, et voici comment :

Code : YAML

```

# Dans un catalogue

charte:
    titre: Mentions légales
    donnee:
        # les éléments de liste sont précédés d'un tiret en YAML
        - >
            Le Site du Zéro recueille des informations (login, e-
            mail) lors de

```

votre enregistrement en tant que membre du site. Lors de votre connexion au site, un fichier "log" stocke les actions effectuées par votre ordinateur (via son adresse IP) au serveur.

- | Lorsque que vous vous connectez en tant que membre du Site du Zéro et que vous cochez la case correspondante, un cookie est envoyé à votre ordinateur afin qu'il se souvienne de votre login et de votre mot de passe. Ceci vous est proposé uniquement afin d'automatiser la procédure de connexion, et n'est en aucun cas utilisé par Simple IT à d'autres fins.

- Merci de votre attention.



On va pouvoir utiliser cela dans une boucle **for** ?

C'est justement l'idée, oui ! On peut utiliser une structure qui va générer une partie de votre page de conditions générales d'utilisation en bouclant sur les valeurs du catalogue, bien vu ! Cela va donner quelque chose comme cela :

Code : HTML & Django

```
{# Dans une vue #}

{%- for i in 0..2 %}
<p>{{ ('charte.donnee.' ~ i )|trans }}</p>
{%- endfor %}
```

La notation `0..2` est une syntaxe Twig pour générer une séquence linéaire. Le nombre avant les deux points (`..`) est le début, celui après est la fin.

Donc quand vous ajoutez un paragraphe, vous l'insérez à la bonne place dans le catalogue, sans vous préoccuper de son numéro. Vous n'avez qu'à incrémenter la fin de la séquence. De même si vous supprimez un paragraphe, vous n'avez qu'à décrémenter la limite de la séquence.



Comme pour PHP, les tableaux récupérés depuis le YAML commencent à 0 et non 1.

Utiliser les domaines

Si vous avez commencé à bien remplir votre fichier `messages.fr.yml`, vous pouvez vous rendre compte qu'il grossit assez vite. Et surtout, qu'il peut y avoir des conflits entre les noms des chaînes sources si vous ne faites pas assez attention.

En fait, il est intéressant de répartir et regrouper les traductions par domaine. Le domaine par défaut est `messages`, c'est pourquoi nous utilisons depuis le début le fichier `messages.XX.XXX`. Un domaine correspond donc à un fichier.

Vous pouvez donc créer autant de fichiers/domaines que vous voulez, la première partie représentant le nom du domaine de traduction que vous devrez utiliser.



Mais comment définir le domaine à utiliser pour telle ou telle traduction ?

C'est un argument à donner à la balise, au filtre ou à la fonction `trans`, tout simplement :

- Balise : `{% trans from 'domaine' %}chaîne{% endtrans %}`.
- Filtre : `{{{ 'chaîne'|trans({}, 'domaine') }}}`.
- Service : `<?php $translator->trans('chaîne', array(), 'domaine')`.

C'est pour cette raison qu'il faut utiliser les domaines avec parcimonie. En effet, si vous décidez d'utiliser un domaine différent de celui par défaut (`messages`), alors il vous faudra le préciser dans chaque utilisation de `trans` ! Attention donc à ne pas créer 50 domaines inutilement, le choix doit avoir un intérêt.

Domaines et bundles

 Quelle est la différence entre un domaine et son bundle ? Est-ce qu'on peut avoir les mêmes domaines dans des bundles différents ?

Autant de questions qui, je le sais, vous taraudent l'esprit. En fait, c'est plutôt simple : les domaines n'ont rien à voir avec les bundles. Voilà, c'est dit.

Du coup, cela veut dire que vous pouvez tout à fait avoir un domaine « A » dans un bundle, et ce même domaine « A » dans un autre bundle. Le contenu de ces deux bouts de catalogue vont s'additionner pour former le catalogue complet du domaine « A ». C'est ce que nous faisons déjà avec le domaine « messages » en fait ! Une vue du bundle « A » pourra alors utiliser une traduction définie dans le bundle « B », et inversement, à condition que le domaine soit le même.

Et si plusieurs fichiers d'un même domaine définissent la même chaîne source, alors c'est le fichier qui est chargé en dernier qui l'emporte (il écrase la valeur définie par les précédents). L'ordre de chargement des fichiers du catalogue est le même que celui de l'instanciation des bundles dans le Kernel. Il faut donc vérifier tout cela dans votre fichier `app/AppKernel.php`.

Un domaine spécial : validators

Vous avez peut-être essayé de traduire les messages que vous affichez lors d'erreurs à la soumission de formulaires, et avez remarqué que vous ne pouviez pas les traduire comme tout le reste.

 Pourtant, les messages d'erreur fournis par le framework étaient traduits, eux !

Oui, mais c'est parce que Symfony2 n'utilise pas le domaine « messages » pour traduire les messages d'erreur des formulaires. Le framework est prévu pour travailler avec le domaine « validators » dans ce contexte des messages d'erreur. Il vous suffit alors de placer vos traductions dans ce domaine (dans le fichier `validators.fr.yml` par exemple), et ce dans le bundle de votre choix comme nous venons de le voir.

Nous reviendrons sur ce domaine spécial un peu plus loin.

Traductions dépendantes de variables

La traduction d'un texte n'est pas quelque chose d'automatique. En effet, toutes les langues ne se ressemblent pas, et il peut y avoir des différences qui ont des conséquences importantes sur notre façon de gérer les traductions.

Prenons deux exemples qui vont vous faire comprendre tout de suite :

- En français, le point d'exclamation est précédé d'une espace, alors qu'en anglais non. Du coup, le « `Hello {{ name }} !` » que l'on a dans notre vue n'est pas bon, car sa traduction devient « `Bonjour {{ name }} !` », sans espace avant le point d'exclamation. La traduction n'est pas correcte !
- En anglais comme en français, mettre au pluriel un nom ne se limite pas toujours à rajouter un « s » à la fin. Comment faire un `if` dans notre vue pour prendre cela en compte ?

Le composant de traduction a tout prévu, ne vous inquiétez pas et regardons cela tout de suite. 😊

Les placeholders



Pour mettre mon espace devant le point d'exclamation français, est-ce que je dois ajouter dans le catalogue la traduction de « ! » en « ! » ?

Bien tenté, mais il y a heureusement une meilleure solution !

La solution apportée par Symfony est relativement simple : on va utiliser un **placeholder**, sorte de paramètre dans une chaîne cible. Cela va nous permettre de régler ce problème d'espacement. Rajoutez ceci dans vos catalogues français et anglais :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/messages.fr.yml
hello: Bonjour %name% !
```

Et

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/messages.en.yml
hello: Hello %name%!
```

Nous avons mis un *placeholder* nommé `%name%` dans chacune des traductions anglaise et française. La valeur de ce placeholder sera spécifiée lors du rendu de la vue, ce qui permet de traduire la phrase complète. Cela évite de découper les traductions avec une partie avant la variable et une partie après la variable, et heureusement lorsque vous avez plusieurs variables dans une même phrase !

Bien entendu il faut adapter un peu notre vue, voici comment passer la valeur du placeholder de la vue au traducteur :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/traduction.html.twig #}
{{ 'hello'|trans({'%name%': name}) }}
```

Le premier paramètre donné ici au filtre `trans` est un tableau, dont les index sont les placeholders avec les caractères `%` qui le délimitent, et les valeurs, celles par lesquelles le placeholder sera remplacé dans la chaîne cible. Nous venons de dire à Symfony que « quand tu traduis la chaîne source "hello", tu vas remplacer `%name%` qui se trouve dans la chaîne cible par le contenu de la variable `name` », qui contient ici le nom de l'utilisateur.



Un placeholder *doit* être encadré par des `%` dans les vues, alors que ce n'est pas réellement nécessaire pour le service. Mais par convention, et pour mieux les voir dans les chaînes cibles lors de l'ajout d'une nouvelle langue par exemple, mieux vaut les utiliser partout.

Du coup, ces caractères `%` doivent être présents dans l'index du tableau des placeholders donné au filtre.

Testez donc l'affichage de cette page en français, puis en anglais. Le point d'exclamation est bien précédé d'une espace en français, mais pas en anglais, et le nom d'utilisateur s'affiche toujours !

Parce qu'on n'utilise pas toujours le filtre, voici les syntaxes pour toutes les possibilités d'utilisation :

- Balise : `{% trans with {'%name%': name} %}hello{% endtrans %}`.
- Filtre : `{{ 'hello'|trans({'%name%': name}) }}`.
- Service : `<?php $translator->trans('hello', array('%name%' => $name)) .`

Et dans le cas où le paramètre a une valeur fixe dans telle vue, vous pouvez bien évidemment utiliser du texte brut à la place du nom de la variable `name`, comme ceci :

Code : Django

```
{% 'hello' | trans({'%name%': 'moi-même'}) %}
```

Les placeholders dans le domaine validators

Les messages d'erreur de formulaires, qui sont donc dans le domaine validators, peuvent contenir des nombres, principalement quand on spécifie des contraintes de longueur. Ces nombres, il faut bien les afficher à l'utilisateur. Pour cela, vous allez me dire qu'il faut utiliser les placeholders.

Raté ! Ce n'est pas du tout comme cela qu'il faut faire dans ce cas. Rassurez-vous, ce n'est que l'exception qui confirme la règle.

Donc dans le cas des messages d'erreur générés par le composant `Validator`, et uniquement dans ce cas, il ne faut pas utiliser les placeholders, mais une syntaxe propre à la validation. Cette syntaxe est la même que celle de Twig en fait : `{ { limit } }`.

Prenons le cas où vous avez utilisé la contrainte `Length`, vous avez envie de mentionner le nombre limite de caractères (que ce soit le maximum ou le minimum) et le nombre de caractères entrés par l'utilisateur. Ces valeurs sont fournies par le service de validation, dans les variables `limit` et `value` respectivement. Ce n'est donc pas `%limit%` qu'il faut utiliser dans votre traduction, mais `{ { limit } }`, comme ceci :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/validators.fr.yml
password:
    length:
        short: "Vous avez entré {{ value }} caractères. Or, le mot de passe ne peut en comporter moins de {{ limit }}"
        long: "Vous avez entré {{ value }} caractères. Or, le mot de passe ne peut en comporter plus de {{ limit }}"
```



Notez les guillemets autour des chaînes cibles. Ils sont aussi à mettre obligatoirement — encore un détail qui vaut une séance chez le coiffeur.

La raison de cette exception est que le validateur n'envoie pas les valeurs de ces variables au traducteur, il les garde pour lui et fait la substitution *après* le retour de la chaîne traduite par le traducteur. Pensez-y !

Gestion des pluriels

On va maintenant essayer d'afficher (et y réussir !) le nombre d'articles correspondant à une catégorie sur votre blog, sous la forme « Il y a (nombre) articles ». Il peut y en avoir un seul ou plusieurs, et comme on veut faire les choses bien, il faut que cela affiche « Il y a 1 article » et « Il y a (plus d'un) articles », avec le « s » qui apparaît quand le nombre d'articles dépasse 1.

Si vous deviez le faire tout de suite, vous feriez sûrement un petit test dans la vue pour choisir quelle chaîne traduire, dans ce style-là :

Code : HTML & Django

```
{# Dans une vue #}
{# Attention, ceci est un mauvais exemple, à ne pas utiliser ! #}
```

```
{% if nombre <= 1 %}
{{ 'article.nombre.singulier'|trans({'%count%': nombre}) }}
{% else %}
{{ 'article.nombre.pluriel'|trans({'%count%': nombre}) }}
{% endif %}
```

Avec le catalogue associé :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/messages.fr.yml
# Attention, ceci est un mauvais exemple, à ne pas utiliser !

article:
    nombre:
        singulier: Il y a %count% article
        pluriel:   Il y a %count% articles
```

Eh bien, votre intention est louable, mais une fois de plus, les concepteurs de Twig et de Symfony ont déjà réfléchi à cela et ont tout prévu ! La nouvelle balise/filtre/fonction à utiliser s'appelle `transchoice`, et elle s'utilise avec en argument le nombre sur lequel faire la condition, voyez par vous-mêmes :

Le filtre :

Code : Django

```
{% 'article.nombre'|transchoice(nombre) %}
```

La balise :

Code : Django

```
{% transchoice nombre %}article.nombre{% endtranschoice %}
```

Le service :

Code : PHP

```
<?php
$translator->transchoice($nombre, 'article.nombre');
```

Le catalogue, quant à lui, contient donc les deux syntaxes dans une même chaîne source. Voici la syntaxe particulière à adopter :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/translations/messages.fr.yml
article:
    nombre: "Il y a %count% article{{'']}{{''}}1,+Inf]Il y a %count% articles"
```

Avec cette syntaxe, Symfony pourra savoir que la première partie est pour 0 ou 1 article, et la seconde pour 2 ou plus. Je ne m'attarderai pas dessus, la [documentation officielle](#) est là si vous voulez absolument plus d'informations.

 Notez que le placeholder `%count%` est automatiquement remplacé par le paramètre donné à la nouvelle fonction `transchoice` pour qu'elle détermine la chaîne cible à utiliser. Il n'est donc pas nécessaire de passer manuellement le tableau de placeholders comme on a pu le faire précédemment. Par contre, le nom du placeholder est obligatoirement `%count%`, il vous faut donc l'utiliser dans la chaîne cible.

Afficher des dates au format local



J'affiche souvent des dates, et j'aimerais avoir les noms des jours/mois, mais comment les traduire ?



Si vous n'avez pas les extensions ICU et intl installées et activées sur votre serveur, la lecture de ce paragraphe ne vous servira à rien.

Vérifiez si votre serveur de production possède ces extensions en accédant au `config.php` disponible dans le répertoire `/web`. Si vous avez une recommandation qui vous parle d'intl, vous devez installer et/ou activer l'extension si vous avez un serveur dédié, et tenter de discuter avec l'hébergeur si vous êtes sur un serveur mutualisé.

Pour afficher les dates sous la forme « vendredi 11 janvier 2013 », vous avez sûrement déjà utilisé le code `{ { date|date('l j F Y') } }`. Malheureusement, l'objet Date de PHP n'est pas très bon en langues... et quelque soit votre locale, les noms de jours et de mois sont en anglais. D'ailleurs, ils le sont même sur [la page de la documentation](#).

Je vous rassure tout de suite : il est bien possible de traduire ces dates ! Dans nos vues Twig, il va falloir pour cela utiliser le filtre `localizeddate` à la place de juste `date`. Son utilisation est la suivante :

Code : HTML & Django

```
{ { date|localizeddate(dateFormat, timeFormat, locale) } }
```

Les paramètres qu'on lui passe sont les suivants :

1. `dateFormat` : le format pour la date ;
2. `timeFormat` : le format pour l'heure ;
3. `locale` : la locale dans laquelle afficher la date formatée. Pas besoin de la spécifier, elle est fournie dans le contexte.



Mais pourquoi séparer les formats de date et d'heure ?

Voilà, c'était trop beau pour être vrai, on ne peut pas utiliser la syntaxe habituelle pour le format de date/heure (du moins, pas encore). À la place, on a le choix entre quatre formats : `full`, `long`, `medium` et `short`, pour l'heure comme pour la date, correspondant aux affichages donnés dans le tableau suivant. Il n'est pas possible de les modifier, mais il est en revanche possible de les combiner (donc avoir la date `long` et l'heure `short`, par exemple). À défaut de pouvoir faire exactement comme vous voulez, vous avez au moins les mois et les jours traduits correctement, et dans un format tout de même convenable. 😊

Formats français de date et heure avec `{ { date|localizeddate(dateFormat, timeFormat, 'fr_FR') } }`

Format	Date	Heure
full	jeudi 15 novembre 2012	14:22:15 Heure normale de l'Europe centrale

long	15 novembre 2012	14:22:15 HNEC
medium	15 nov. 2012	14:22:15
short	15/11/12	14:22
none	(rien)	(rien)

Notez que si la locale est simplement `fr`, une virgule s'ajoute après le nom du jour (donnant « jeudi, 15 novembre 2012 ») ainsi qu'un « `h` » après la chaîne « `H:m:s` » (« 14:22:15 h Heure normale de l'Europe centrale ») pour le format `full`, et la date au format `short` aura pour séparateurs des points au lieu de slashes. On n'utilise que très rarement les formats `full` et `long` pour l'heure. 😊

J'ai précisé ici en dur la locale, mais dans votre code ne la mettez pas : elle est automatiquement définie à la locale courante. Votre utilisation sera ainsi aisée :

Code : Django

```
Aujourd'hui nous sommes le {{ 'now'|localizeddate('full', 'none') }}
et il est {{ 'now'|localizeddate('none', 'short') }}
```

Et si vous l'exécutez en même temps que j'écris ces lignes (ce qui me paraît impossible...), vous obtiendrez :

Citation : Résultat

Aujourd'hui nous sommes le lundi 14 janvier 2013 et il est 20:02

Attention, si vous rencontrez l'erreur suivante : « `The filter "localizeddate" does not exist in ...` », c'est que vous n'avez pas encore activé l'extension Twig qui fournit ce filtre. Pour cela, rajoutez simplement cette définition de service dans votre fichier de configuration :

Code : YAML

```
# app/config/config.yml

# ...
# Activation de l'extension Twig intl
services:
    twig.extension.intl:
        class: Twig_Extensions_Extension_Intl
        tags:
            - { name: twig.extension }
```

Pour information, les autres extensions Twig peuvent se trouver à l'adresse suivante : [https://github.com/fabpot/Twig-extensi \[...\] ons/Extension](https://github.com/fabpot/Twig-extensi [...] ons/Extension).

Pour conclure

Voici pour terminer un petit récapitulatif des différentes syntaxes complètes, sachant que la plupart des arguments sont facultatifs.

Les balises :

Code : HTML & Django

```

{# Texte simple #}
{%
    trans with {'%placeholder%': placeholderValue} from 'domaine'
    into locale %}maChaine{%
    endtrans %}

{# Texte avec gestion de pluriels #}
{%
    transchoice count with {'%placeholder%': placeholderValue} from
    'domaine' into locale %}maChaine{%
    endtranschoice %}

```

Les filtres :

Code : HTML & Django

```

{# Texte simple #}
{{ 'maChaine'|trans({'%placeholder%': placeholderValue}, 'domaine',
locale) }}

{# Texte avec gestion de pluriels #}
{{ 'maChaine'|transchoice (count, {'%placeholder%':
placeholderValue}, 'domaine', locale) }}

```

Les méthodes du service :

Code : PHP - Service

```

<?php
$translator = $this->get('translator'); // depuis un contrôleur

// Texte simple
$translator->trans('maChaine', array('%placeholder%' =>
$placeholderValue), 'domaine', $locale);

// Texte avec gestion de pluriels
$translator->transchoice($count, 'maChaine', array('%placeholder%' =>
$placeholderValue), 'domaine', $locale)

```

Vous savez maintenant comment créer les traductions dans les différentes langues que vous souhaitez gérer sur votre site !

En résumé

- La méthodologie d'une traduction est la suivante :
 - Détermination du texte à traduire : cela se fait grâce à la balise et au filtre Twig, ou directement grâce au service translator ;
 - Détermination de la langue cible : cela s'obtient avec la locale, que Symfony2 définit soit à partir de l'URL, soit à partir des préférences de l'internaute.
- Traduction à l'aide d'un dictionnaire : cela correspond aux catalogues dans Symfony ;
- Il existe plusieurs formats possibles pour les catalogues, le YAML étant le plus simple ;
- Il existe différentes méthodes pour bien organiser ses catalogues, pensez-y !
- Il est possible de faire varier les traductions en fonction de paramètres et/ou de pluriels.

Partie 5 : Astuces et points particuliers

Cette partie recense différentes astuces et points particuliers de Symfony2, qui vous permettent de réaliser des choses précises dans votre projet.

Utiliser des ParamConverters pour convertir les paramètres de requêtes

L'objectif des *ParamConverters*, ou « convertisseurs de paramètres », est de vous faire gagner du temps et des lignes de code. Sympa, non ? 😊

Il s'agit de transformer automatiquement un paramètre de route, comme `{id}` par exemple, en un objet, une entité `$article` par exemple. Vous ne pourrez plus vous en passer ! Et bien entendu, il est possible de créer vos propres convertisseurs, qui n'ont de limite que votre imagination. 😊

Théorie : pourquoi un ParamConverter ?

Récupérer des entités Doctrine avant même le contrôleur

Sur la page d'affichage d'un article de blog, par exemple, n'êtes-vous pas fatigués de toujours devoir vérifier l'existence de l'article demandé, et de l'instancier vous-mêmes ? N'avez-vous pas l'impression d'écrire toujours et encore les mêmes lignes ?

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/ArticleController.php

// ...

public function voirAction($id)
{
    $em      = $this->getDoctrine()->getManager();
    $article = $em->find('Sdz\BlogBundle\Entity\Article', $id);

    if (null !== $article) {
        throw $this->createNotFoundException('L\'article demandé [id='.$id.'] n\'existe pas.');
    }

    // Ici seulement votre vrai code...

    return $this->render('SdzBlogBundle:Blog:voir.html.twig',
array('article' => $article));
}
```

Pour enfin vous concentrer sur votre code métier, Symfony2 a évidemment tout prévu !

Les ParamConverters

Vous pouvez créer ou utiliser des ParamConverters qui vont agir juste avant le contrôleur. Comme son nom l'indique, un ParamConverter *convertit* les paramètres de votre route au format que vous préférez. En effet, depuis la route, vous ne pouvez pas tellement agir sur vos paramètres. Tout au plus, vous pouvez leur imposer des contraintes via des expressions régulières. Les ParamConverters pallient cette limitation en agissant après le routeur pour venir transformer à souhait ces paramètres.

Le résultat des ParamConverters est stocké dans les attributs de requête, c'est-à-dire qu'on peut les injecter dans les arguments de l'action du contrôleur.

Un ParamConverter utile : DoctrineParamConverter

Vous l'aurez deviné, ce ParamConverter va nous convertir nos paramètres directement en entités Doctrine ! L'idée est la suivante : dans le contrôleur, au lieu de récupérer le paramètre de route `{id}` sous forme de variable `$id`, on va récupérer directement une entité Article sous la forme d'une variable `$article`, qui correspond à l'article portant l'id `$id`.

Et un bonus en prime : on veut également que, s'il n'existe pas d'article portant l'id `$id` dans la base de données, alors une exception 404 soit levée. Après tout, c'est comme si l'on mettait dans la route : `requirements: Article exists!`

Un peu de théorie sur les ParamConverters



Comment fonctionne un ParamConverter ?

Un ParamConverter est en réalité un simple *listener*, qui écoute l'évènement `kernel.controller`.

On l'a vu dans le chapitre sur les évènements, cet évènement est déclenché lorsque le noyau de Symfony2 sait quel contrôleur exécuter (après le routeur, donc), mais avant d'exécuter effectivement le contrôleur. Ainsi, lors de cet évènement, le ParamConverter va lire la signature de la méthode du contrôleur pour déterminer le type de variable que vous voulez. Cela lui permet de créer un attribut de requête du même type, à partir du paramètre de la route, que vous récupérez ensuite dans votre contrôleur.

Pour déterminer le type de variable que vous voulez, le ParamConverter a deux solutions. La première consiste à regarder la signature de la méthode du contrôleur, c'est-à-dire le typage que vous définissez pour les arguments :

Code : PHP

```
<?php  
public function testAction(Article $article)
```

Ici, le typage est Article, devant le nom de la variable. Le ParamConverter sait alors qu'il doit créer une entité Article.

La deuxième solution consiste à utiliser une annotation `@ParamConverter`, ce qui nous permet de définir nous-mêmes les informations dont il a besoin.

Au final, depuis votre contrôleur, vous avez en plus du paramètre original de la route un nouvel argument créé par votre ParamConverter qui s'est exécuté avant votre contrôleur. Et bien entendu, il sera possible de créer vos propres ParamConverters !



Pratique : utilisation des ParamConverters existants

Utiliser le ParamConverter Doctrine

Ce ParamConverter fait partie du bundle `Sensio\FrameworkBundle`. C'est un bundle activé par défaut avec la distribution standard de Symfony2, que vous avez si vous suivez ce cours depuis le début.

Vous pouvez donc vous servir du `DoctrineParamConverter`. Il existe plusieurs façons de l'utiliser, avec ou sans expliciter l'annotation. Voyons ensemble les différentes méthodes.

1. S'appuyer sur l'id et le typage de l'argument

C'est la méthode la plus simple, et peut-être la plus utilisée. Imaginons que vous ayez cette route :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml  
  
sdzblog_voir:  
    path: /blog/article/{id}
```

```
defaults: { _controller: SdzBlogBundle:Blog:voir }
requirements:
    id: \d+
```

Une route somme toute classique, dans laquelle figure un paramètre `{ id }`. On a mis une contrainte pour que cet id soit un nombre, très bien. Le seul point important est que le paramètre s'appelle « `id` », ce qui est aussi le nom d'un attribut de l'entité `Article`.

Maintenant, la seule chose à changer pour utiliser le `DoctrineParamConverter` est côté contrôleur, où il faut typer un argument de la méthode, comme ceci :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php
use Sdz\BlogBundle\Entity\Article;

public function voirAction($id, Article $article)
{
    // Ici, $article est une instance de l'entité Article, portant
    l'id $id
}
```

Faites le test ! Vous verrez que `$article` est une entité pleinement opérationnelle. Vous pouvez l'afficher, créer un formulaire avec, etc. Bref, vous venez d'économiser le `$em->find()` nécessaire pour récupérer manuellement l'entité !

De plus, si vous mettez dans l'URL un id qui n'existe pas, alors le `DoctrineParamConverter` vous lèvera une exception, résultant en une page d'erreur 404 comme dans la figure suivante.



i Ici j'ai laissé l'argument `$id` dans la définition de la méthode, mais vous pouvez tout à fait l'enlever. Vu qu'on a l'article dans la variable `$article`, `$id` n'est plus utile, on peut (et on doit) utiliser `$article->getId()`. C'était juste pour vous montrer que le `ParamConverter` crée un attribut de requête, sans toucher à ceux existants.

i Quand je parle d'attributs de requête, ce sont les attributs que vous pouvez récupérer de cette manière : `$request->attributes->get('article')`. Ce sont ces attributs que vous pouvez injecter dans le contrôleur en tant qu'arguments de la méthode (dans l'exemple précédent, il s'agit de `$id` et `$article`). Mais ce n'est en rien obligatoire, si vous ne les injectez pas ils seront tout de même attributs de la requête.

Avec cette méthode, la seule information que le `ParamConverter` utilise est le typage d'un argument de la méthode, et non le nom de l'argument. Par exemple, vous pourriez tout à fait avoir ceci :

Code : PHP

```
<?php
public function voirAction(Article $bidule)
```

Cela ne change en rien le comportement, et la variable `$bidule` contiendra une instance de l'entité `Article`.

Une dernière note sur cette méthode. Ici cela a fonctionné car le paramètre de la route s'appelle « `id` », et que l'entité `Article` a un attribut `id`. En fait, cela fonctionne avec tous les attributs de l'entité `Article` ! Appelez votre paramètre de route `titre`, et accédez à une URL de type `/blog/article/titre-existant`, cela fonctionne exactement de la même manière ! Cependant, pour l'utilisation d'autres attributs que `id`, je vous conseille d'utiliser les méthodes suivantes.

2. Utiliser l'annotation pour faire correspondre la route et l'entité

Il s'agit maintenant d'utiliser explicitement l'annotation de `DoctrineParamConverter`, afin de personnaliser au mieux le comportement. Considérons maintenant que vous avez la route suivante :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
sdzblog_voir:
    path:      /blog/article/{article_id}
    defaults: { _controller: SdzBlogBundle:Blog:voir }
```

La seule différence est que le paramètre de la route s'appelle maintenant « `article_id` ». On aurait pu tout aussi bien l'appeler « `bidule` », l'important est que ce soit un nom qui n'est pas également un attribut de l'entité `Article`. Le `ParamConverter` ne peut alors pas faire la correspondance automatiquement, il faut donc le lui dire.

Cela ne nous fait pas peur ! Voici l'annotation à utiliser :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/**
 * @ParamConverter("article", options={"mapping": {"article_id": "id"}})
 */
public function voirAction(Article $article)
```

Il s'agit maintenant d'être un peu plus rigoureux. Dans l'annotation `@ParamConverter`, voici ce qu'il faut renseigner :

- Le premier argument de l'annotation correspond au nom de l'argument de la méthode que l'on veut injecter. Le `article` de l'annotation correspond donc au `$article` de la méthode.
- Le deuxième argument correspond aux options à passer au `ParamConverter`. Ici nous avons passé une seule option `mapping`. Cette option fait la correspondance « paramètre de route » => « attribut de l'entité ». Dans notre exemple, c'est ce qui permet de dire au `ParamConverter` : « le paramètre de route `article_id` correspond à l'attribut `id` de l'`Article` ».

Le `ParamConverter` connaît le type d'entité à récupérer (`Article`, `Categorie`, etc.) en lisant, comme précédemment, le typage de l'argument.

Bien entendu, il est également possible de récupérer une entité grâce à plusieurs attributs. Prenons notre entité

ArticleCompetence par exemple, qui est identifiée par deux attributs : article et competence. Il suffit pour cela de passer les deux attributs dans l'option mapping de l'annotation, comme suit :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

// La route serait par exemple : /blog/{article_id}/{competence_id}

/**
 * @ParamConverter("articleCompetence", options={"mapping": {"article_id": "article", "competence_id": "competence"}})
 */
public function voirAction(ArticleCompetence $articleCompetence)
```

3. Utiliser les annotations sur plusieurs arguments

Grâce à l'annotation, il est alors possible d'appliquer plusieurs ParamConverters à plusieurs arguments. Prenez la route suivante :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_voir:
    path:
        /blog/article/{article_id}/commentaires/{commentaire_id}
        defaults: { _controller: SdzBlogBundle:Blog:voir }
```

L'idée ici est d'avoir deux paramètres dans la route, qui vont nous permettre de récupérer deux entités grâce au ParamConverter.

Vous l'aurez compris, avec deux paramètres à convertir il vaut mieux tout expliciter grâce aux annotations, plutôt que de reposer sur une devinette. 😊 La mise en application est très simple, il suffit de définir deux annotations, chacune très simple comme on l'a déjà vu. Voici comment le faire :

Code : PHP

```
<?php
/**
 * @ParamConverter("article", options={"mapping": {"article_id": "id}})
 * @ParamConverter("commentaire", options={"mapping": {"commentaire_id": "id}})
 */
public function voirAction(Article $article, Commentaire
$commentaire)
```

Utiliser le ParamConverter Datetime

Ce ParamConverter est plus simple : il se contente de convertir une date d'un format défini en un objet de type Datetime. Très pratique !

Partons donc de cette route par exemple :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
sdzblog_liste:
    path:      /blog/{date}
    defaults: { _controller: SdzBlogBundle:Blog:voirListe }
```

Et voici comment utiliser le convertisseur sur la méthode du contrôleur :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/**
 * @ParamConverter("date", options={"format": "Y-m-d"})
 */
public function voirListeAction(\Datetime $date)
```

Ainsi, au lieu de simplement recevoir l'argument `$date` qui vaut « 2012-09-19 » par exemple, vous récupérez directement un objet `Datetime` à cette date, vraiment sympa.



Attention, ce ParamConverter fonctionne différemment de celui de Doctrine. En l'occurrence, il ne *crée pas* un nouvel attribut de requête, mais *remplace* l'existant. La conséquence est que le nom de l'argument (ici, `$date`) *doit* correspondre au nom du paramètre dans la route (ici, `{date}`).

Aller plus loin : créer ses propres ParamConverters

Comment sont exécutés les ParamConverters ?

Avant de pouvoir créer notre ParamConverter, étudions comment ils sont réellement exécutés.

À l'origine de tout, il y a un *listener*, il s'agit de

`Sensio\Bundle\FrameworkExtraBundle\Event\Listener\ParamConverterListener`. Ce *listener* écoute l'événement `kernel.controller`, ce qui lui permet de connaître le contrôleur qui va être exécuté. L'idée est qu'il parcourt les différents ParamConverters pour exécuter celui qui convient le premier. On peut synthétiser son comportement par le code suivant :

Code : PHP

```
<?php

foreach ($converters as $converter) {
    if ($converter->supports($configuration)) {
        if ($converter->apply($request, $configuration)) {
            return;
        }
    }
}
```



Vous n'avez pas à écrire ce code, je vous le donne uniquement pour que vous compreniez le mécanisme interne !

Dans ce code :

- La variable `$converters` contient la liste de tous les ParamConverters, nous voyons plus loin comment elle est construite ;
- La méthode `$converter->supports()` demande au ParamConverter si le paramètre actuel l'intéresse ;
- La variable `$configuration` contient les informations de l'annotation : le typage de l'argument, les options de l'annotation, etc. ;
- La méthode `$converter->apply()` permet d'exécuter à proprement parler le ParamConverter.

L'ordre des convertisseurs est donc très important, car si le premier retourne `true` lors de l'exécution de sa méthode `apply()`, alors les éventuels autres ne seront pas exécutés.

Comment Symfony2 trouve tous les convertisseurs ?

Pour connaître tous les convertisseurs, Symfony2 utilise un mécanisme que nous avons déjà utilisé : les tags des services. Vous l'aurez compris, un convertisseur est avant tout un service, sur lequel on a appliqué un tag.

Commençons donc par créer la définition d'un service, que nous allons implémenter en tant que ParamConverter :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdzblog.paramconverter_test
        class: Sdz\BlogBundle\ParamConverter\TestParamConverter
        tags:
            - { name: request.param_converter, priority: 20 }
```

On a ajouté un tag `request.param_converter` sur notre service, ce qui permet de l'enregistrer en tant que tel. J'ai mis ici une priorité de 20, histoire de passer avant le ParamConverter de Doctrine. À vous de voir si vous voulez passer avant ou après, suivant les cas.

Créer un convertisseur

Créons maintenant la classe du ParamConverter. Un convertisseur doit implémenter l'interface [ParamConverterInterface](#).

Commençons par créer la classe d'un convertisseur `TestParamConverter` sur ce squelette, que je place dans le répertoire `ParamConverter` du bundle :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/ParamConverter/TestParamConverter.php

namespace Sdz\BlogBundle\ParamConverter;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ConfigurationInterface;
use Sensio\Bundle\FrameworkExtraBundle\Request\ParamConverter\ParamConverterInterface;
use Symfony\Component\HttpFoundation\Request;

class TestParamConverter implements ParamConverterInterface
{
    function supports(ConfigurationInterface $configuration)
    {

    }

    function apply(Request $request, ConfigurationInterface $configuration)
    {
    }
}
```



L'interface ne définit que deux méthodes : `supports()` et `apply()`.

La méthode `supports()`

La méthode `supports()` doit retourner `true` lorsque le convertisseur souhaite convertir le paramètre en question, `false` sinon. Les informations sur le paramètre courant sont stockées dans l'argument `$configuration`, et contiennent :

- `$configuration->getClass()` : le typage de l'argument dans la méthode du contrôleur ;
- `$configuration->getName()` : le nom de l'argument dans la méthode du contrôleur ;
- `$configuration->getOptions()` : les options de l'annotation, si elles sont explicitées (vide bien sûr lorsqu'il n'y a pas l'annotation).

Vous devez, avec ces trois éléments, décider si oui ou non le convertisseur compte convertir le paramètre.

La méthode `apply()`

La méthode `apply()` doit effectivement créer un attribut de requête, qui sera injecté dans l'argument de la méthode du contrôleur.

Ce travail peut être effectué grâce à ses deux arguments :

- La configuration, qui contient les informations sur l'argument de la méthode du contrôleur, que nous avons vu juste au-dessus ;
- La requête, qui contient tout ce que vous savez, et notamment les paramètres de la route courante via `$request->attributs->get('paramètre_de_route')`.

L'exemple de notre `TestParamConverter`

Histoire de bien comprendre ce que chaque méthode et chaque variable doit faire, je vous propose un petit exemple. Imaginons que notre site internet est disponible via différents noms de domaine. Dans une certaine action de contrôleur, on veut récupérer une entité `Site`, dont l'attribut `hostname` correspond au nom de domaine courant.

Du coup, notre `TestParamConverter` ne se sert même pas d'un des paramètres de la route courante. Il ne va se servir que du nom de domaine contenu directement dans la requête. C'est pourquoi la route n'a rien de particulier, elle peut tout à fait ne contenir aucun paramètre.

Le service

Tout d'abord, on a besoin de l'`EntityManager` dans notre service, afin de pouvoir récupérer l'entité `Site` qui nous intéresse. On va également passer en paramètre le nom de la classe de l'entité `Site` en question. Normalement vous savez le faire, voici comment je l'ai réalisé à partir de la définition du service qu'on a écrite plus haut :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/services.yml

services:
    sdzblob.paramconverter_test
        class: Sdz\BlogBundle\ParamConverter\TestParamConverter
        arguments: ['Sdz\BlogBundle\Entity\Site',
        @doctrine.orm.entity_manager]
        tags:
            - { name: request.param_converter, priority: 20 }
```

La classe

Ensuite, il faut modifier le squelette du convertisseur qu'on a réalisé plus haut. Voici mon implémentation :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/ParamConverter/TestParamConverter.php

namespace Sdz\BlogBundle\ParamConverter;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ConfigurationInterface;
use Sensio\Bundle\FrameworkExtraBundle\Request\ParamConverter\ParamConverterInterface;
use Symfony\Component\HttpFoundation\Request;
use Doctrine\ORM\EntityManager;

class TestParamConverter implements ParamConverterInterface
{
    protected $class;
    protected $repository;

    public function __construct($class, EntityManager $em)
    {
        $this->class = $class;
        $this->repository = $em->getRepository($class);
    }

    function supports(ConfigurationInterface $configuration)
    {
        // $conf->getClass() contient la classe de l'argument dans la méthode du
        // contrôleur
        // On teste donc si cette classe correspond à notre classe Site, contenue
        // dans $this->class
        return $configuration->getClass() == $this->class;
    }

    function apply(Request $request, ConfigurationInterface $configuration)
    {
        // On récupère l'entité Site correspondante
        $site = $this->repository->findOneByHostname($request->getHost());

        // On définit ensuite un attribut de requête du nom de $conf->getName()
        // et contenant notre entité Site
        $request->attributes->set($configuration->getName(), $site);

        // On retourne true pour qu'aucun autre ParamConverter ne soit utilisé sur
        // cet argument
        // Je pense notamment au ParamConverter de Doctrine qui risque de vouloir
        // s'appliquer !
        return true;
    }
}
```

Le contrôleur

Pour utiliser votre convertisseur flambant neuf, il ne faut pas grand-chose. Comme je vous l'ai mentionné précédemment, on n'utilise pas ici de paramètre venant de la route. L'utilisation est donc très simple : il faut juste ajouter un argument, typé en Site, comme ceci :

Code : PHP

```
<?php  
// src/Sdz/BlogBundle/Controller/BlogController.php  
  
use Sdz\BlogBundle\Entity\Site;  
  
public function indexAction($page, Site $site)  
{  
    // Ici, $site est une instance de Site avec comme hostname «  
    // localhost » (si vous testez en local !)  
}
```

Pas besoin d'expliciter l'annotation ici, car nous n'avons aucune option particulière à passer au ParamConverter. Mais si vous en aviez, c'est le seul moyen de le faire donc gardez-le en tête. 😊

Maintenant, à chaque fois que vous avez besoin d'accéder à l'entité Site correspondant au nom de domaine sur lequel vous êtes, il vous suffira de rajouter un argument Site \$site dans la méthode de votre contrôleur. C'est tout ! Le ParamController s'occupe du reste.

 Bien sûr, mon exemple est incomplet. Il faudrait une gestion des erreurs, notamment lorsqu'il n'existe pas d'entité Site avec pour hostname le domaine sur lequel vous êtes. Il suffirait pour cela de déclencher une exception NotFoundHttpException. Je vous invite à vous inspirer de ceux existants : DoctrineParamConverter et DatetimeParamConverter

- Un ParamConverter vous permet de créer un attribut de requête, que vous récupérez ensuite en argument de vos méthodes de contrôleur ;
- Il existe deux ParamConverters par défaut avec Symfony2 : Doctrine et Datetime ;
- Il est facile de créer ses propres convertisseurs pour accélérer votre développement.

Personnaliser les pages d'erreur

Avec Symfony2, lorsqu'une exception est déclenchée, le noyau l'attrape. Cela lui permet ensuite d'effectuer l'action adéquate.

Le comportement par défaut du noyau consiste à appeler un contrôleur particulier intégré à Symfony2 :

`TwigBundle\Exception:show`. Ce contrôleur récupère les informations de l'exception, choisit le template adéquat (un template différent par type d'erreur : 404, 500, etc.), passe les informations au template et envoie la réponse générée.

À partir de là, il est facile de personnaliser ce comportement : `TwigBundle` étant... un bundle, on peut le modifier pour l'adapter à nos besoins ! Mais ce n'est pas le comportement que nous voulons changer, c'est juste l'apparence de nos pages d'erreur. Il suffit donc de créer nos propres templates et de dire à Symfony2 d'utiliser nos templates et non ceux par défaut.

Théorie : remplacer les vues d'un bundle

Constater les pages d'erreur

Les pages d'erreur de Symfony2 sont affichées lorsque le noyau attrape une exception. Il existe deux pages différentes : celle en mode `dev` et celle en mode `prod`.

Il est possible de personnaliser les deux, mais celle qui nous intéresse le plus ici est la page d'erreur en mode production. En effet, c'est celle qui sera affichée à nos visiteurs ; elle mérite donc toute notre attention.

Je vous invite donc à vous remémorer ce à quoi elle ressemble. Pour cela, accédez à une URL inexistante via `app.php`, et voyez le résultat à la figure suivante.

Oops! An Error Occurred

The server returned a "404 Not Found".

Une page d'erreur

Something is broken. Please e-mail us at [email] and let us know what you were doing when this error occurred. We will fix it as soon as possible. Sorry for any inconvenience caused.

pas très séduisante

Comme vous pouvez le constater, ce n'est pas très présentable pour nos futurs visiteurs !

Localiser les vues concernées

Avant de chercher à modifier des vues, il faut d'abord bien les identifier. Comme je vous l'ai précisé, les vues de ces pages d'erreur se situent dans le bundle `TwigBundle`, et plus précisément dans le répertoire `vendor\symfony\symfony\src\Symfony\Bundle\TwigBundle\Resources\views\Exception`.



Je vous donne leur localisation pour information, mais n'allez surtout pas modifier directement les fichiers de ce répertoire ! Comme tout ce qui se situe dans le répertoire `vendor`, ils sont susceptibles d'être écrasés par Composer à la prochaine mise à jour.

Remplacer les vues d'un bundle

Il est très simple de remplacer les vues d'un bundle quelconque par les nôtres. Il suffit de créer le répertoire `app/Resources/NomDuBundle/views/` et d'y placer nos vues à nous ! Et cela est valable quelque soit le bundle.

Nos vues doivent porter exactement les mêmes noms que celles qu'elles remplacent. Ainsi, si notre bundle utilise une vue située dans :

Code : Autre

```
... (namespace) /RépertoireDuBundle/Resources/views/Hello/salut.html.twig
```

alors nous devons créer la vue :

Code : Autre

```
app/Resources/NomDuBundle/views/Hello/salut.html.twig
```



Attention, NomDuBundle correspond bien au nom du bundle, à savoir au nom du fichier que vous pouvez trouver à sa racine. Par exemple : SdzBlogBundle est le nom du bundle, mais il se trouve dans (src)/Sdz/BlogBundle.

La figure suivante présente un schéma pour bien comprendre, appliqué à la vue error.html.twig du bundle TwigBundle.

Ancienne vue :

.../TwigBundle/Resources/views/Exception/error.html.twig

Nouvelle vue :

Nom du Bundle

app/Resources/TwigBundle/views/Exception/error.html.twig

Syntaxe pour remplacer une vue

Comportement de Twig

Twig, pour chaque vue qu'on lui demande de retourner, regarde d'abord dans le répertoire app/Resources s'il trouve la vue correspondante. S'il ne la trouve pas, il va ensuite voir dans le répertoire du bundle.

Ainsi, ici pour chaque TwigBundle:Exception:error.html.twig, Twig ira vérifier dans le répertoire app avant de prendre la vue du bundle TwigBundle.



Attention, ceci n'est valable que pour les vues, car c'est le comportement de Twig. Cela ne fonctionne pas pareil pour tout ce qui est contrôleur et autres !

Pourquoi il y a tous les formats error.XXX.twig dans le répertoire Exception ?

C'est une très bonne question, et si vous les ouvrez vous vous rendrez compte que chaque vue d'erreur est compatible au format de son extension. Cela permet de ne pas générer des erreurs en cascade.

Je m'explique, imaginons que vous chargez un fichier JS, généré dynamiquement par l'un de vos contrôleurs (pourquoi pas !). Si ce contrôleur génère une erreur quelconque, et qu'il affiche cette erreur en HTML, alors votre navigateur qui attend du JavaScript sera perdu ! Il va tenter d'exécuter le retour du contrôleur en tant que JavaScript, mais le retour est en réalité du HTML et générera donc pas mal d'erreurs dans votre navigateur.

C'est pour éviter ce comportement que Symfony2 fournit plusieurs formats d'erreur. Ainsi, si le format de votre réponse est défini comme du JavaScript, alors Symfony2 utilisera la vue error.js.twig qui est, si vous l'ouvrez, compatible JavaScript car en commentaire. Vous remercierez Symfony2 la prochaine fois que cela vous arrivera !

Pratique : remplacer les templates Exception de TwigBundle

Créer la nouvelle vue

Maintenant qu'on sait le faire, il ne reste plus qu'à le faire ! Créez donc le répertoire

app/Resources/TwigBundle/views/Exception. Et au sein de ce répertoire, le bundle utilise la convention suivante pour chaque nom de template :

- Il vérifie d'abord l'existence de la vue error[code_erreur].html.twig, par exemple error404.html.twig dans le cas d'une page introuvable (erreur 404) ;
- Si ce template n'existe pas, il utilise la vue error.html.twig, une sorte de page d'erreur générique.

Je vous conseille de créer un error404.html.twig pour les pages non trouvées, en plus du error.html.twig générique. Cela vous permet d'afficher un petit texte sympa pour que l'utilisateur ne soit pas trop perdu.

Le contenu d'une page d'erreur

Pour savoir quoi mettre dans ces vues, je vous propose de jeter un œil à celle qui existe déjà, error.html. Vous la trouvez comme indiqué plus haut dans le répertoire

vendor/symfony/src/Symfony/Bundle/TwigBundle/Resources/views/Exception, voici son contenu :

Code : HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>An Error Occurred: {{ status_text }}</title>
  </head>
  <body>
    <h1>Oops! An Error Occurred</h1>
    <h2>The server returned a "{{ status_code }} {{ status_text }}".</h2>

    <div>
      Something is broken. Please e-mail us at [email] and let us know what you were doing when this error occurred. We will fix it as soon as possible. Sorry for any inconvenience caused.
    </div>
  </body>
</html>
```

Vous pouvez y voir les différentes variables que vous pouvez utiliser : {{ status_text }} et {{ status_code }}. Fort de cela, vous pouvez créer la page d'erreur que vous souhaitez : vous avez toutes les clés.



Je le rappelle : cette page d'erreur que l'on vient de personnaliser, c'est la page d'erreur générée en mode prod !



Remplacer la page d'erreur du mode dev n'a pas beaucoup d'intérêt : vous seuls la voyez, et elle est déjà très complète. Cependant, si vous souhaitez quand même la modifier, alors cela n'est pas le template error.html.twig qu'il faut créer, mais le template exception.html.twig. Celui-ci se trouve aussi dans le répertoire Exception.

En résumé

- Modifier les vues d'un bundle quelconque est très pratique, votre site garde ainsi une cohérence dans son design, et ce, que ce soit sur votre bundle à vous comme sur les autres ;
- Personnaliser les pages d'erreur, ce n'est pas la priorité lorsque l'on démarre un projet Symfony2, mais c'est impératif avant de l'ouvrir à nos visiteurs.

Utiliser Assetic pour gérer les codes CSS et JS de votre site

La gestion des ressources CSS et JavaScript dans un site internet est très importante et n'est pas si évidente ! Leur chargement est très souvent le point le plus lent pour l'affichage de la page à vos visiteurs, ce n'est donc pas quelque chose à négliger.

Pour vous aider à gérer ces ressources efficacement, Symfony2 intègre un bundle nommé Assetic qui va s'occuper de tout cela à votre place. Il va vous permettre d'optimiser au maximum le chargement de ces ressources pour vos visiteurs. Vous verrez, ce bundle est presque magique !

Théorie : entre vitesse et lisibilité, pourquoi choisir ?

À propos du nombre de requêtes HTTP d'une page web

Vous devez sûrement vous demander ce que vient faire la vitesse d'une page dans une section qui traite de code CSS. C'est une bonne question et je vais y répondre. Le temps de chargement ressenti d'une page par un visiteur comprend tout le processus du clic au rendu de la page par le navigateur. Ainsi, on y inclut :

- Le temps d'envoi de la requête au serveur lors du clic. On ne peut pas y faire grand-chose, malheureusement.
- Le temps d'exécution de la page côté serveur, le temps PHP, donc. Pour cela, il faut bien penser son script et essayer de l'optimiser un peu.
- Le temps d'envoi du code HTML par le serveur vers le navigateur. On ne peut pas y faire grand-chose non plus.
- Mais là ce n'est pas tout : à partir de cette page HTML que le navigateur reçoit, ce dernier doit *tout* recommencer pour chaque fichier CSS, chaque JavaScript et chaque image !

Donc si votre page contient 5 fichiers CSS, 3 JavaScript et 15 images, cela fait un total de 23 requêtes HTTP à traiter par votre navigateur pour vous afficher l'intégralité de la page ! Et pour ces 23 requêtes, il y a les temps d'envoi et de réception qui sont incompressibles et qui prennent du temps.

Au final, s'il faut bien sûr optimiser le code PHP côté serveur, la partie *front-end* qui comprend codes HTML, CSS et JavaScript ainsi que les fichiers images est bien celle qui prend le plus de temps à se charger, vu du visiteur.

Comment optimiser le *front-end* ?

L'idée est de réduire les temps incompressibles. Comme ils sont justement incompressibles, il faut que l'on en diminue le nombre. La seule solution est donc de grouper ces fichiers. L'idée est que, au lieu d'avoir cinq fichiers CSS différents, on va mettre tout notre code CSS dans un seul fichier. Comme cela, on aura une seule requête au lieu de cinq. Super !

Mais le problème, c'est que si l'on avait trois fichiers et non un seul, ce n'était pas pour rien. Chaque fichier concernait une partie de votre site, c'était bien plus lisible. Tout regrouper vous gênerait dans le développement de vos fichiers CSS (*idem* pour les fichiers JavaScript, bien sûr).

C'est là qu'Assetic va intervenir : il va grouper lui-même les fichiers et va vous permettre de garder votre séparation !

Il est aussi possible d'améliorer le temps de chargement !

En effet, transmettre votre unique fichier CSS de plusieurs centaines de lignes, cela prend du temps (temps qui varie en fonction de la connexion de votre serveur, de celle du visiteur, etc.). On peut améliorer ce temps en diminuant simplement la taille du fichier.

C'est possible grâce à un outil Java appelé *YUI Compressor*, un outil développé par Yahoo!. Cet outil permet de diminuer la taille de vos fichiers CSS, mais surtout de vos fichiers JavaScript, en supprimant les commentaires, les espaces, en raccourcissant le nom des variables, etc. On dit qu'il « minifie » les fichiers (il ne les compresse pas comme un fichier zip). Le code devient bien sûr complètement illisible ! Mais c'est là qu'Assetic intervient de nouveau : il vous permet de garder votre version claire lorsque vous développez, mais « minifie » les fichiers pour vos visiteurs (en mode prod) !

Conclusion

Grâce à Assetic, on peut optimiser très facilement nos scripts CSS/JS. Par exemple, nous pouvons passer de nos huit requêtes pour 500 Ko à seulement deux requêtes (1 CSS + 1 JS) pour 200 Ko. Le temps d'affichage de la page pour nos visiteurs sera donc

bien plus court, et on conserve la lisibilité du code côté développeur !

Pratique : Assetic à la rescousse !

Assetic est déjà activé par défaut dans Symfony2, donc vous n'avez rien à faire de ce côté, vous pouvez l'utiliser directement. L'objectif d'Assetic est de regrouper nos fichiers CSS et JavaScript comme nous venons d'en parler.

Servir des ressources

Assetic peut servir au navigateur les ressources que vous lui demandez.

Servir une seule ressource

Allez donc dans la vue du layout, nous allons y déclarer nos fichiers CSS et JavaScript. Voici comment on déclarait nos fichiers CSS jusqu'à maintenant :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

<link rel="stylesheet" href="{{ asset('bundles/sdzblog/css/main.css') }}" type="text/css" />
```

Et voici comment faire pour décharger cette responsabilité à Assetic :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #

{% stylesheets '@SdzBlogBundle/Resources/public/css/main.css' %}
  <link rel="stylesheet" href="{{ asset_url }}" type="text/css" />
{%- endstylesheets %}
```



Au sein de la balise `stylesheets`, la variable `{{ asset_url }}` est définie, et vaut l'URL à laquelle le navigateur peut récupérer le CSS.

Et voici le HTML qu'Assetic a généré avec cette balise :

Code : HTML

```
<link rel="stylesheet"
href="/Symfony/web/app_dev.php/css/519c4f6_main_1.css"
type="text/css" />
```

Pas convaincant ? C'est parce que nous sommes en mode dev ! Nous verrons plus loin comment nous occuper du mode prod, qui demande un peu d'effort.

En attendant, essayons de comprendre ce code généré. En mode dev, Assetic génère à la volée les ressources, d'où une URL vers un fichier CSS qui passe par le contrôleur frontal `app_dev.php`. En réalité, c'est bien un contrôleur d'Assetic qui s'exécute, car le fichier `app_dev.php/css/519c4f6_main_1.css` n'existe évidemment pas. Ce contrôleur va chercher le contenu du fichier qu'on lui a indiqué, puis le retransmet. Pour l'instant il le retransmet tel quel, mais il sera bien sûr possible d'appliquer des modifications, nous le verrons par la suite.

Et bien sûr, le mécanisme est exactement le même pour vos fichiers JavaScript :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

{%- javascripts '@SdzBlogBundle/Resources/public/js/main.js' %}
<script type="text/javascript" src="{{ asset_url }}></script>
{%- endjavascripts %}
```

 Pensez bien à mettre les fichiers JavaScript en toute fin du code HTML. C'est parce que lorsque le navigateur intercepte une déclaration de fichier JavaScript, il arrête le rendu de la page, charge le fichier, l'exécute, et ensuite seulement continue le rendu de la page. En effet, le navigateur ne peut pas savoir d'avance si le script JavaScript veut changer quelque chose à l'apparence de la page, il est donc obligé de tout bloquer avant de continuer. Si vos scripts JavaScript sont bien faits, vous pouvez sans problème les déclarer en fin de page. Ainsi, le navigateur se bloquera pour vos fichiers JavaScript une fois que le rendu de la page sera terminé ! Ce qui fait que c'est transparent pour vos visiteurs : ils peuvent déjà profiter de la page pendant que les scripts JS se chargent en arrière-plan.

Servir plusieurs ressources regroupées en une

Cela devient déjà un peu plus intéressant. En plus du fichier CSS `main.css` (ou tout autre fichier, adaptez au code que vous avez bien sûr), on va charger le fichier `bootstrap.css` qui est directement dans `web/css`. Avec l'ancienne méthode, on aurait écrit une deuxième balise `<link>`, mais voici comment faire avec Assetic :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

{%- stylesheets
  '@SdzBlogBundle/Resources/public/css/main.css'
  'css/bootstrap.css' %}
<link rel="stylesheet" href="{{ asset_url }}" type="text/css" />
{%- endstylesheets %}
```

On a simplement rajouté la deuxième ressource à charger dans la balise `stylesheets`. Et voici le rendu HTML :

Code : HTML

```
<link rel="stylesheet"
href="/Symfony2/web/app_dev.php/css/03b7e21_main_1.css"
type="text/css" />
<link rel="stylesheet"
href="/Symfony2/web/app_dev.php/css/03b7e21_bootstrap_2.css"
type="text/css" />
```

 Mais il n'était pas censé regrouper les deux ressources en une ?

Si bien sûr... mais en mode prod ! Encore une fois, nous sommes en mode de développement, il est donc inutile de regrouper les ressources (on se fiche un peu de la rapidité), Assetic ne le fait donc pas.

Si vous avez plusieurs fichiers CSS dans le répertoire des CSS de votre bundle, il est également possible d'utiliser un joker pour les charger tous. Ainsi, au lieu de préciser les fichiers exacts :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

{%
    stylesheets
    '@SdzBlogBundle/Resources/public/css/main.css'
    '@SdzBlogBundle/Resources/public/css/autre.css' %}
}
```

Vous pouvez utiliser le joker « * », comme ceci :

Code : Django

```
{# app/Resources/views/layout.html.twig //}

{%
    stylesheets '@SdzBlogBundle/Resources/public/css/*' %}
```

Ce qui chargera tous les fichiers qui sont dans le répertoire, pratique !



Avec cette méthode, vous constatez qu'Assetic va chercher les ressources directement dans les répertoires du bundle. Vous n'avez donc plus à publier les ressources publiques de votre bundle dans le répertoire web. 😊

Modifier les ressources servies

En servant les ressources depuis un contrôleur PHP, Assetic a la possibilité de modifier à la volée tout ce qu'il sert. Cela est possible grâce aux filtres, que l'on peut définir directement dans les balises `stylesheets` ou `javascripts`.

Voyons quelques filtres intéressants.

Le filtre `cssrewrite`

Si vous avez exécuté le code précédent, vous avez dû vous rendre compte qu'il se pose un petit problème : les images utilisées par le CSS de bootstrap ont disparu. En effet, le fichier CSS de bootstrap fait référence aux images via le chemin relatif `../img/exemple.png`.

Lorsque le fichier CSS était placé dans `web/css`, ce chemin relatif pointait bien vers `web/img`, là où sont nos images. Or maintenant, *du point de vue du navigateur*, le fichier CSS est dans `app_dev.php/css`, du coup le chemin relatif vers les images n'est plus bon !

C'est ici qu'intervient le filtre `cssrewrite`. Voici la seule modification à apporter côté vue Twig :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

{%
    stylesheets filter='cssrewrite'
    '@SdzBlogBundle/Resources/public/css/main.css'
    'css/bootstrap.css' %}
    <link rel="stylesheet" href="{{ asset_url }}" type="text/css" />
{%
    endstylesheets %}
```

On a juste précisé l'attribut `filter` à la balise. Ce filtre permet de réécrire tous les chemins relatifs contenus dans les fichiers CSS, afin de prendre en compte la modification du répertoire du CSS. Actualisez votre page, vous verrez que cela fonctionne très bien ! Le chemin relatif d'accès aux images est devenu : `../img/exemple.png`, ce qui est bon.

Les filtres yui_css et yui_js

Ces filtres sont très utiles, ce sont ceux qui « minifient » les fichiers avec *YUI Compressor*.

Pour utiliser l'outil YUI Compressor, il faut que vous [le téléchargez](#) manuellement. Copiez le fichier (version 2.4.8-pre à l'heure où j'écris ces lignes) dans le répertoire app/Resources/java, par exemple. Maintenant, direction la configuration de notre application, il y a une section sur Assetic pour lui dire où nous avons mis Yui Compressor. Modifiez la partie assetic :

Code : YAML

```
# app/config/config.yml

# Assetic Configuration
assetic:
    debug:          %kernel.debug%
    use_controller: false
    # java: /usr/bin/java
    filters:
        cssrewrite: ~
        yui_js:
            jar: %kernel.root_dir%/Resources/java/yuicompressor.jar
        yui_css:
            jar: %kernel.root_dir%/Resources/java/yuicompressor.jar
```



Le %kernel.root_dir% représente le répertoire de votre application, à savoir le répertoire app.



Si vous êtes sous Windows ou que votre exécutable Java ne se trouve pas dans /usr/bin/java, il faut alors décommenter le paramètre java dans la configuration d'Assetic. Changez sa valeur en C:\Program Files (x86)\Java\jre7\bin\java.exe (valeur par défaut pour Windows 8 64 bits), ou toute autre valeur qui convient à votre configuration (à vous de voir où est votre fichier java.exe).

Voilà, nous venons d'activer les filtres yui_js et yui_css, on peut maintenant les utiliser depuis nos vues. Ajoutez ce filtre dans vos balises :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

{%- stylesheets filter='cssrewrite, yui_css'
... %}
```

Et de même pour les fichiers JavaScript :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

{%- javascripts filter='yui_js'
... %}
```

Testez le rendu !

Mais... on est toujours en mode dev et les fichiers sont devenus illisibles pour un éventuel débogage ! Heureusement, vous

avez la possibilité de dire qu'un filtre ne s'applique pas en mode dev. Il suffit de mettre un point d'interrogation devant :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #}

{%
    stylesheets filter='?yui_css'
    ...
%}
```

Ainsi, le filtre ne s'appliquera qu'en mode prod, tout comme le regroupement des fichiers en un seul.

Au final, notre mode dev n'a pas changé d'un poil, on garde nos différents fichiers et ces derniers sont lisibles, mais le mode prod a reçu toutes les optimisations : regroupement des fichiers ainsi que « minification ».

Gestion du mode prod

Si vous n'avez pas encore testé le rendu en mode prod, faites-le. Cela ne fonctionne pas ? Vos fichiers CSS et JS ne sont pas chargés ? C'est normal. 😊 Nous n'avons pas fini notre mise en place.

Comprendre Assetic

Pour comprendre pourquoi la gestion du mode prod demande un effort supplémentaire, vous devez comprendre la manière dont Assetic fonctionne. Lorsque l'on utilise les balises `{% stylesheets %}` ou `{% javascripts %}`, le code HTML généré en mode prod est le suivant (regardez la source de vos pages HTML) :

Code : HTML

```
<link rel="stylesheet" href="/Symfony2/web/css/cd91cad.css"
      type="text/css" />
```

Or ce fichier n'existe pas du tout !

Lors du mode dev, on l'a vu, Assetic passe directement par un contrôleur pour générer à la volée nos ressources. Mais évidemment, « minifier » et regrouper des fichiers à la volée et ce pour chaque requête, cela prend beaucoup de temps. Si en mode dev on peut se le permettre, on ne le peut pas en mode prod !

Du coup, l'astuce pour le mode prod est d'exporter en dur, une bonne fois pour toutes, les fichiers CSS et JS dont on a besoin. Ainsi, en mode prod, le fichier `/css/cd91cad.css` (dans mon cas) existera en dur, Assetic n'interceptera pas l'URL, et votre serveur web (souvent Apache) va envoyer directement le contenu du fichier à vos visiteurs. Plus rapide, on ne peut pas !



Vous notez que, maintenant que nous sommes en mode prod, Assetic a bien regroupé toutes nos ressources en une seule.

Exporter ses fichiers CSS et JS

Pour faire cet export en dur, il faut utiliser une simple commande d'Assetic :

Code : Console

```
php app/console assetic:dump --env=prod
```

Cette commande devrait vous sortir un résultat de ce type :

Code : Console

```
C:\wamp\www\Symfony>php app/console assetic:dump --env=prod
Dumping all prod assets.
Debug mode is off.

16:13:30 [file+] C:/wamp/www/Symfony/app/../web/css/cd91cad.css
```

Cette commande va lire toutes nos vues pour y trouver les balises `{% stylesheets %}` et `{% javascripts %}`, puis va exporter en dur dans les fichiers `/web/css/XXX.css` et `/web/js/XXX.js`.

Et voilà, maintenant, nos fichiers existent réellement. Testez à nouveau le rendu en mode prod : c'est bon !



Attention : exporter nos fichiers en dur est une action ponctuelle. Ainsi, *à chaque fois que vous modifiez vos fichiers d'origine, vous devez exécuter la commande assetic:dump pour mettre à jour vos fichiers pour la production !* Prenez donc l'habitude, à chaque fois que vous déployez en production, de vider le cache et d'exporter les ressources Assetic.

Et bien plus encore...

Assetic, c'est une bibliothèque complète qui permet beaucoup de choses. Vous pouvez également optimiser vos images, et construire une configuration plus poussée. Bref, n'hésitez pas à vous renseigner sur [la documentation officielle](#).

- Le chargement des fichiers CSS et JS prend beaucoup de temps dans le rendu d'une page HTML sur le navigateur de vos visiteurs ;
- Assetic permet de regrouper tous vos fichiers CSS ainsi que tous vos fichiers JS, afin de réduire le nombre de requêtes HTTP que doivent faire vos visiteurs pour afficher une page ;
- Assetic permet également de minifier vos fichiers, afin de diminuer leur taille et donc accélérer leur chargement pour vos visiteurs ;
- Enfin, l'utilisation d'Assetic permet de garder votre confort de développement en local, vos fichiers ne sont ni regroupés, ni minifiés : indispensable pour déboguer !

Utiliser la console directement depuis le navigateur

La console est un outil bien pratique de Symfony2. Mais parfois, devoir ouvrir le terminal de Linux ou l'invite de commandes de Windows n'est pas très agréable. Et je ne parle pas des hébergements mutualisés, qui n'offrent pas d'accès SSH pour utiliser la console !

Comment continuer d'utiliser la console dans ces conditions ? Ce chapitre est là pour vous expliquer cela !

Théorie : le composant Console de Symfony2

Les commandes sont en PHP

Nous l'avons déjà évoqué au cours de ce tutoriel, les commandes Symfony2 sont bien de simples codes PHP ! Effectivement on les exécute depuis une console, mais cela ne les empêche en rien d'être en PHP.

Et comme elles sont en PHP... elle peuvent tout à fait être exécutées depuis un autre script PHP. C'est en fait ce qui est déjà fait par le script PHP de la console, celui que l'on exécute à chaque fois : le fichier `app/console`. Voici son contenu :

Code : PHP

```
#!/usr/bin/env php
<?php

require_once __DIR__ . '/bootstrap.php.cache';
require_once __DIR__ . '/AppKernel.php';

use Symfony\Bundle\FrameworkBundle\Console\Application;
use Symfony\Component\Console\Input\ArgvInput;

$input = new ArgvInput();
$env = $input->getParameterOption(array('--env', '-e'),
getenv('SYMFONY_ENV') ?: 'dev');
$debug = !$input->hasParameterOption(array('--no-debug', ''));

$kernel = new AppKernel($env, $debug);
$application = new Application($kernel);
$application->run();
```

Comme vous pouvez le voir, ce fichier ressemble beaucoup au contrôleur frontal, `app.php`. Il charge également le Kernel. La seule chose qu'il fait de différent, c'est d'utiliser le composant Console de Symfony2, en instanciant la classe `Application` (ligne 15). C'est cet objet qui va ensuite exécuter les différentes commandes définies en PHP dans les bundles.

Exemple d'une commande

Chaque commande est définie dans une classe PHP distincte, que l'on place dans le répertoire `Command` des bundles. Ces classes comprennent entre autres deux méthodes :

- `configure()` qui définit le nom, les arguments et la description de la commande ;
- `execute()` qui exécute la commande à proprement parler.

Prenons l'exemple de la commande `list`, qui liste toutes les commandes disponibles dans l'application. Elle est définie dans le fichier `vendor/symfony/src/Component/Console/Command>ListCommand.php`, dont voici le contenu :

Code : PHP

```
<?php

/*
 * This file is part of the Symfony package.
 *
 * (c) Fabien Potencier <fabien@symfony.com>
```

```
*  
* For the full copyright and license information, please view the  
LICENSE  
* file that was distributed with this source code.  
*/  
  
namespace Symfony\Component\Console\Command;  
  
use Symfony\Component\Console\Input\InputArgument;  
use Symfony\Component\Console\Input\InputOption;  
use Symfony\Component\Console\Input\InputInterface;  
use Symfony\Component\Console\Output\OutputInterface;  
use Symfony\Component\Console\Output\Output;  
use Symfony\Component\Console\Command\Command;  
  
/**  
 * ListCommand displays the list of all available commands for the  
application.  
*  
* @author Fabien Potencier <fabien@symfony.com>  
*/  
class ListCommand extends Command  
{  
    /**  
     * {@inheritDoc}  
     */  
    protected function configure()  
    {  
        $this  
            ->setDefinition(array(  
                new InputArgument('namespace', InputArgument::OPTIONAL, 'The  
namespace name'),  
                new InputOption('xml', null, InputOption::VALUE_NONE, 'To  
output help as XML'),  
            ))  
            ->setName('list')  
            ->setDescription('Lists commands')  
            ->setHelp(<<<EOF  
The <info>list</info> command lists all commands:  
  
<info>php app/console list</info>  
  
You can also display the commands for a specific namespace:  
  
<info>php app/console list test</info>  
  
You can also output the information as XML by using the <comment>--  
xml</comment> option:  
  
<info>php app/console list --xml</info>  
EOF  
        );  
    }  
  
    /**  
     * {@inheritDoc}  
     */  
    protected function execute(InputInterface $input, OutputInterface  
$output)  
    {  
        if ($input->getOption('xml')) {  
            $output->writeln($this->getApplication()->asXml($input-  
>getArgument('namespace')), OutputInterface::OUTPUT_RAW);  
        } else {  
            $output->writeln($this->getApplication()->asText($input-  
>getArgument('namespace')));  
        }  
    }  
}
```

Vous distinguez bien ici les deux méthodes qui composent la commande `list`. En vous basant sur cet exemple, vous êtes d'ailleurs capables d'écrire votre propre commande : ce n'est vraiment pas compliqué !

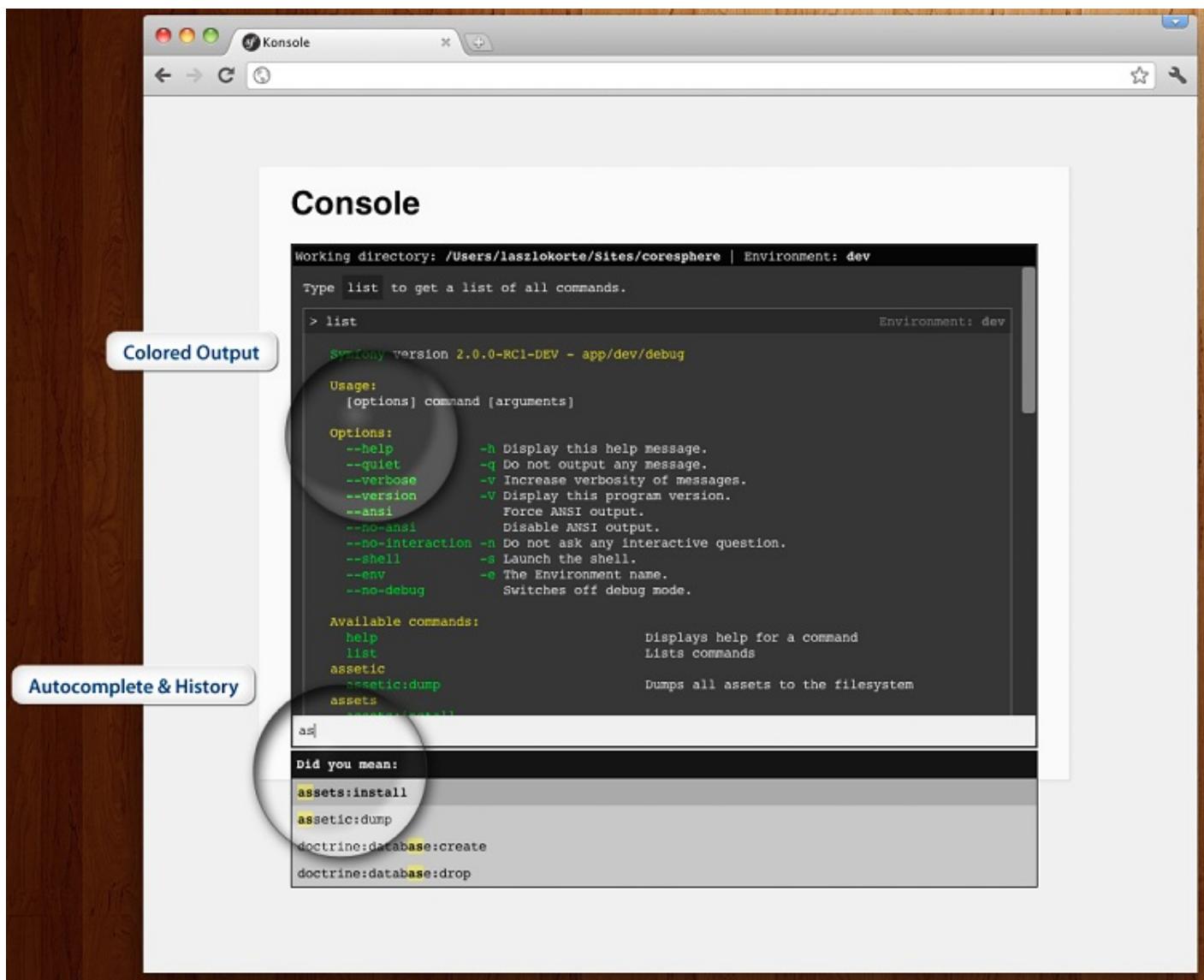
Mais revenons au but de ce chapitre, qui est de pouvoir utiliser ces commandes depuis le navigateur.

Pratique : utiliser un **ConsoleBundle**

ConsoleBundle ?

Vous le savez sûrement, la communauté de Symfony2 est très active, et un nombre impressionnant de bundles a vu le jour depuis la sortie de Symfony2. Vous pouvez les retrouver presque tous sur le site <http://knxbundles.com/> qui les recense.

Il doit sûrement y avoir plusieurs bundles qui fournissent une console dans le navigateur, mais je vous propose d'en installer un en particulier : `CoreSphereConsoleBundle`. C'est un bundle simple qui remplit parfaitement sa tâche, et dont l'interface est très pratique, comme le montre la figure suivante.



Interface de `CoreSphereConsoleBundle`

Installer **CoreSphereConsoleBundle**

L'installation d'un tel bundle est vraiment simple, attaquons-la dès maintenant.

Télécharger `CoreSphereConsoleBundle`

Pour installer ce bundle, je vous propose de télécharger une version que j'ai modifiée. J'ai entre autres résolu quelques petits

bugs et traduit les messages en français. L'adresse du bundle est donc la suivante :
<https://github.com/winzou/ConsoleBundle>.

Pour l'installer avec Composer, rajoutez la ligne suivante dans vos dépendances :

Code : JavaScript

```
// composer.json

"require": {
    // ...
    "winzou/console-bundle": "dev-master"
}
```

Puis mettez à jour vos dépendances grâce à la commande suivante :

Code : Console

```
php ../composer.phar update
```

Si jamais vous n'avez pas Composer, ce que je déconseille, vous pouvez le télécharger à la main depuis GitHub. Cliquez sur Download et téléchargez l'archive (le format zip est recommandé pour les utilisateurs de Windows). Décompressez ensuite l'archive dans le répertoire vendor/bundles/CoreSphere/ConsoleBundle. Ensuite, enregistrez le namespace dans votre autoload comme ceci :

Code : PHP

```
<?php
// app/autoload.php

// ...

$loader->add('CoreSphere', __DIR__.'/../vendor/bundles');

return $loader;
```

Enregistrement du bundle dans le Kernel

Puis il faut enregistrer le bundle CoreSphereConsoleBundle dans app/AppKernel.php (ligne 33) :

Code : PHP

```
<?php

use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new Symfony\Bundle\SecurityBundle\SecurityBundle(),
            new Symfony\Bundle\TwigBundle\TwigBundle(),
            new Symfony\Bundle\MonologBundle\MonologBundle(),
            new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),

```

```

    new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
    new Symfony\Bundle\AsseticBundle\AsseticBundle(),
    new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
    new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),

    // D'autres bundles que vous auriez déjà pu ajouter
);

if (in_array($this->getEnvironment(), array('dev', 'test'))) {
    $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
    $bundles[] = new
    Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
    $bundles[] = new
    Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
    $bundles[] = new
    Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();

    // D'autres bundles que vous auriez déjà pu ajouter

    // On enregistre ce bundle uniquement pour l'environnement de
    développement évidemment
    $bundles[] = new
    CoreSphere\ConsoleBundle\CoreSphereConsoleBundle();
}

return $bundles;
}

// ...
}

```

Enregistrement des routes

Pour paramétrier un bundle, on fait comme toujours : on lit sa documentation. La documentation se trouve soit dans le `readme`, soit dans le répertoire `Resources/doc`, cela dépend des bundles. Dans notre cas, elle se trouve dans le `readme`.

Pour les routes, il faut donc enregistrer le fichier dans notre `routing_dev.yml`. On ne les met pas dans `routing.yml`, car la console ne doit être accessible qu'en mode dev, on a enregistré le bundle que pour ce mode. Ajoutez donc à la fin de `app/config/routing_dev.yml` :

Code : Autre

```

console:
    resource: "@CoreSphereConsoleBundle/Resources/config/routing.yml"

```

Publier les assets

L'installation touche à sa fin, il ne reste plus qu'à rendre disponibles les fichiers JS et CSS du bundle, ce qui se fait comme vous le savez grâce à la commande suivante :

Code : Console

```
php app/console assets:install --symlink web
```

C'est fini ! Il ne reste plus qu'à utiliser notre nouvelle console.



Bien sûr, lorsque je vous dit d'exécuter cette commande, c'est en local ! Exécutez-la sur votre PC, puis envoyez tous les fichiers par FTP sur votre serveur, vous aurez ainsi accès à la console sur votre serveur. 😊

Utilisation de la console dans son navigateur

Par défaut, le bundle définit la route `/console` pour afficher la console. Allez donc à l'adresse http://localhost/Symfony/web/app_dev.php/console et profitez !



Bien entendu, pour exécuter des commandes Symfony2 depuis cette interface, il ne faut pas faire `php app/console la_commande`, mais uniquement `la_commande` ! Le script PHP du bundle n'utilise pas le script `app/console`, il utilise directement le composant `Console`.

Pour les utilisateurs de Windows, vous pouvez remarquer que le résultat des commandes est en couleurs. Eh oui, Symfony2 est plus fort que l'invite de commandes de Windows, il gère les couleurs !

En plus de l'adresse `/console` dédiée, j'ai rajouté un petit bouton console, regardez en bas à droite dans la barre d'outils de Symfony. Cliquez dessus, et une petite console s'ouvre par-dessus votre page. Pratique pour exécuter une commande rapidement ! Pour enlever la console, cliquez de nouveau sur le bouton.

Prêts pour l'hébergement mutualisé

Vous êtes prêts pour utiliser la console de votre application sur les hébergements mutualisés, qui n'offrent généralement pas d'accès SSH !

En résumé

- Les commandes Symfony2 sont en PHP pur, il est ainsi tout à fait possible de « simuler » une console via le navigateur.
- Vous disposez maintenant d'une console accessible depuis votre navigateur : cela va vous simplifier la vie, croyez-moi ! 😊
- N'hésitez pas à faire vos retours sur le bundle directement via les *issues* sur GitHub : <https://github.com/winzou/ConsoleBundle/issues>.

Déployer son site Symfony2 en production

Votre site est fonctionnel ? Il marche parfaitement en local, et vous voulez que le monde entier en profite ? Vous êtes au bon endroit, on va voir dans ce chapitre les points à vérifier pour déployer votre site sur un serveur distant.

L'objectif de ce chapitre n'est pas de vous apprendre comment mettre en production un site de façon générale, mais juste de vous mettre le doigt sur les quelques points particuliers auxquels il faut faire attention lors d'un projet **Symfony2**.

La méthodologie est la suivante :

1. Uploader votre code à jour sur le serveur de production ;
2. Mettre à jour vos dépendances via Composer ;
3. Mettre à jour votre base de données ;
4. Vider le cache.

Préparer son application en local

Bien évidemment, la première chose à faire avant d'envoyer son application sur un serveur, c'est de bien vérifier que tout fonctionne chez soi ! Vous êtes habitués à travailler dans l'environnement de développement et c'est normal, mais pour bien préparer le passage en production, on va maintenant utiliser le mode production.

Vider le cache, tout le cache

Tout d'abord, pour être sûrs de tester ce qui est codé, il faut vider le cache. Faites donc un petit :

Code : Console

```
php app/console cache:clear
```

Voici qui vient de vider le cache... de l'environnement de développement ! Eh oui, n'oubliez donc jamais de bien vider le cache de production, via la commande :

Code : Console

```
php app/console cache:clear --env=prod
```

Tester l'environnement de production

Pour tester que tout fonctionne correctement en production, il faut utiliser le contrôleur frontal `app.php` comme vous le savez, et non `app_dev.php`. Mais cet environnement n'est pas très pratique pour détecter et résoudre les erreurs, vu qu'il ne les affiche pas du tout. Pour cela, ouvrez le fichier `web/app.php`, on va activer le mode *debugger* pour cet environnement. Il correspond au deuxième argument du constructeur du Kernel :

Code : PHP

```
<?php  
// web/app.php  
  
// ...  
  
$kernel = new AppKernel('prod', true); // Définissez ce 2e argument  
à true
```

Dans cette configuration, vous êtes toujours dans l'environnement de production, avec tous les paramètres qui vont bien : rappelez-vous, certains fichiers comme `config.yml` ou `config_dev.yml` sont chargés différemment selon l'environnement. L'activation du mode *debugger* ne change rien à cela, mais permet d'afficher à l'écran les erreurs.



Pensez à bien remettre ce paramètre à `false` lorsque vous avez fini vos tests !



Lorsque le mode `debugger` est désactivé, les erreurs ne sont certes pas affichées à l'écran, mais elles sont heureusement répertoriées dans le fichier `app/logs/prod`. Si l'un de vos visiteurs vous rapporte une erreur, c'est dans ce fichier qu'il faut aller regarder pour avoir le détail, les informations nécessaires à la résolution de l'erreur.

Soigner ses pages d'erreur

En tant que développeurs, vous avez la chance de pouvoir utiliser l'environnement de développement et d'avoir de très jolies pages d'erreur, grâce à Symfony2. Mais mettez-vous à la place de vos visiteurs : créez volontairement une erreur sur l'une de vos pages (une fonction Twig mal orthographiée par exemple), et regardez le résultat depuis l'environnement de production (et sans le mode `debugger` bien sûr !), visible à la figure suivante.

Oops! An Error Occurred

The server returned a "404 Not Found".

Une page d'erreur

Something is broken. Please e-mail us at [email] and let us know what you were doing when this error occurred. We will fix it as soon as possible. Sorry for any inconvenience caused.

pas très séduisante

Pas très joli, n'est-ce pas ? C'est pour cela qu'il faut impérativement que vous personnalisiez les pages d'erreur de l'environnement de production. Un chapitre entier est dédié à ce point important, je vous invite à lire « [Personnaliser les pages d'erreur](#) ».

Installer une console sur navigateur

En fonction de l'hébergement que vous avez, vous n'avez pas forcément l'accès SSH nécessaire pour exécuter les commandes Symfony2. Heureusement, les commandes Symfony2 sont de simples scripts PHP, il est alors tout à fait possible de les exécuter depuis un navigateur. Il existe des bundles qui émulent une console dans un navigateur, décrits dans un chapitre dédié : je vous invite à lire le chapitre « [Utiliser la console directement depuis le navigateur](#) ».

Vérifier et préparer le serveur de production

Vérifier la compatibilité du serveur

Évidemment, pour déployer une application Symfony2 sur votre serveur, encore faut-il que celui-ci soit compatible avec les besoins de Symfony2 ! Pour vérifier cela, on peut distinguer deux cas.

Vous avez déjà un hébergeur

Ce cas est le plus simple, car vous avez accès au serveur. Vous le savez, Symfony2 intègre un petit fichier PHP qui fait toutes les vérifications de compatibilité nécessaires, utilisons-le ! Il s'agit du fichier `web/config.php`, mais avant de l'envoyer sur le serveur il nous faut le modifier un petit peu. En effet, ouvrez-le, vous pouvez voir qu'il y a une condition sur l'IP qui appelle le fichier :

Code : PHP

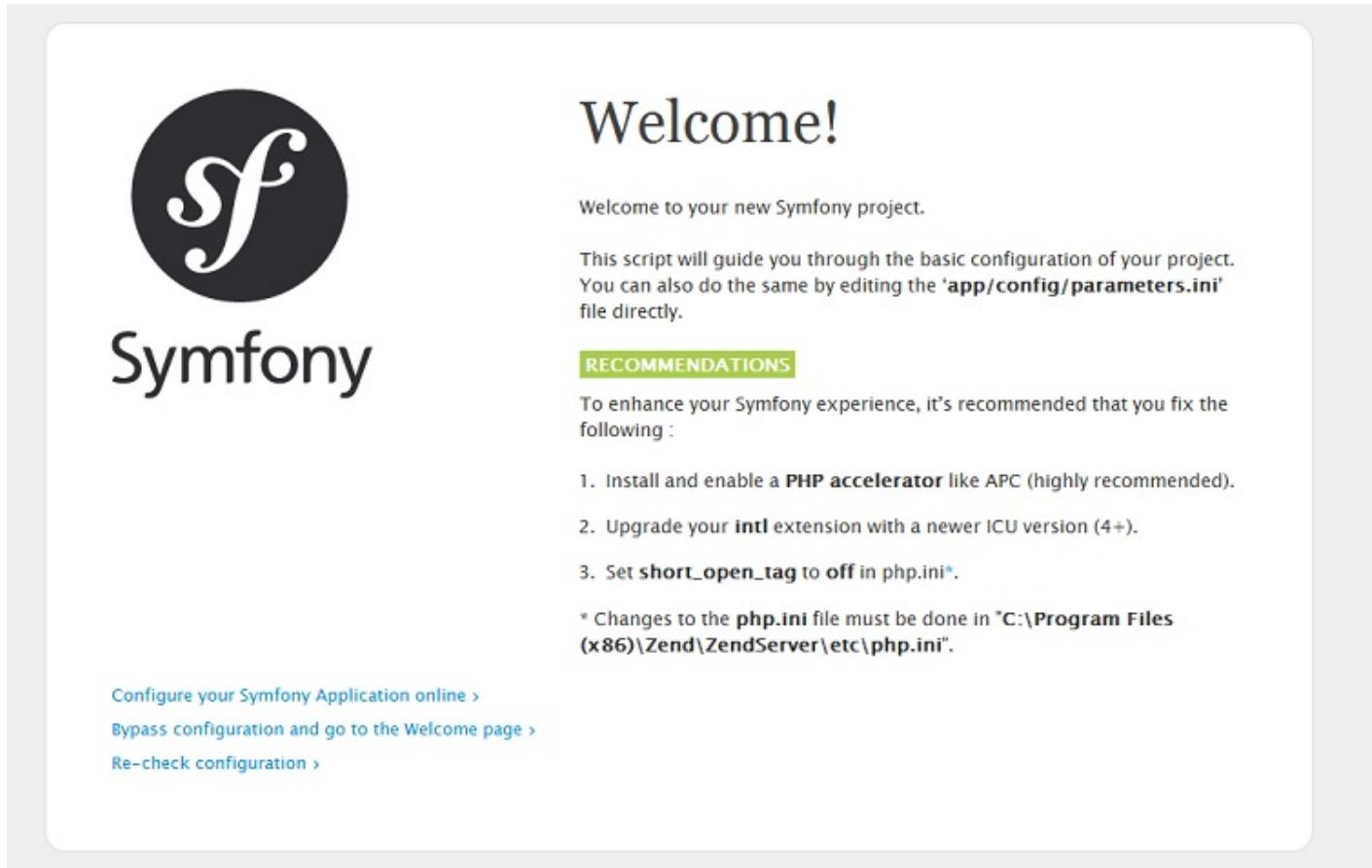
```
<?php  
// web/config.php  
// ...
```

```

if (!in_array(@$_SERVER['REMOTE_ADDR'], array(
    '127.0.0.1',
    '::1',
))) {
    header('HTTP/1.0 403 Forbidden');
    exit('This script is only accessible from localhost.');
}

```

Comme ce fichier n'est pas destiné à rester sur votre serveur, supprimez simplement ce bloc et envoyez le fichier sur votre serveur. Ouvrez la page web qui lui correspond, par exemple www.votre-serveur.com/config.php. Vous devriez obtenir la figure suivante.



Le fichier de configuration s'affiche

Comme vous le voyez, mon serveur est compatible avec Symfony2, car il n'y a pas de partie « Major Problems », juste des « Recommendations ». Bien évidemment, essayez de respecter les recommandations avec votre hébergeur/administrateur si cela est possible. Notamment, comme Symfony2 l'indique, installer un accélérateur PHP comme APC est très important, cela augmentera très sensiblement les performances. Si celles-ci n'étaient pas importantes en local, elles le seront en ligne !



Si vous avez envoyé seulement le fichier `config.php`, vous aurez bien sûr les deux problèmes majeurs comme quoi Symfony2 ne peut pas écrire dans les répertoires `app/cache` et `app/logs`. Pas d'inquiétude, on enverra tous les autres fichiers un peu plus tard.

Vous n'avez pas encore d'hébergeur et en cherchez un compatible

Dans ce cas, vous ne pouvez pas exécuter le petit script de test inclus dans Symfony2. Ce n'est pas bien grave, vous allez le faire à la main ! Voici les points obligatoires qu'il faut que votre serveur respecte pour pouvoir faire tourner Symfony2 :

- La version de PHP doit être supérieure ou égale à PHP 5.3.3 ;
- L'extension SQLite 3 doit être activée ;

- L'extension JSON doit être activée ;
- L'extension Ctype doit être activée ;
- Le paramètre date.timezone doit être défini dans le php.ini.

Il y a bien entendu d'autres points qu'il vaut mieux vérifier, bien qu'ils ne soient pas obligatoires. La liste complète est disponible dans [la documentation officielle](#).

Modifier les paramètres OVH pour être compatible

Certains hébergeurs permettent la modification de certains paramètres via les .htaccess ou l'interface d'administration. Il m'est bien sûr impossible de lister toutes les solutions pour chaque hébergement. C'est pourquoi ce paragraphe est uniquement à destination des personnes hébergées chez OVH, il y en a beaucoup et c'est un cas un peu particulier.

Vous savez sans doute que le PHP par défaut d'OVH est une branche de la version 4, or Symfony2 a besoin de la version 5.3.2 minimum. Pour cela, créez un fichier .htaccess à la racine de votre hébergement, dans le répertoire www :

Code : Autre

```
SetEnv SHORT_OPEN_TAGS 0
SetEnv REGISTER_GLOBALS 0
SetEnv MAGIC_QUOTES 0
SetEnv SESSION_AUTOSTART 0
SetEnv ZEND_OPTIMIZER 1
SetEnv PHP_VER 5_3
```

Ceci permettra notamment d'activer la version 5.3 de PHP, mais également de définir quelques autres valeurs utiles au bon fonctionnement de Symfony2.

Déployer votre application

Envoyer les fichiers sur le serveur

Dans un premier temps, il faut bien évidemment envoyer les fichiers sur le serveur. Pour éviter d'envoyer des fichiers inutiles et lourds, videz dans un premier temps le cache de votre application : celui-ci est de l'ordre de 1 à 10 Mo. Attention, pour cette fois il faut le vider à la main, en supprimant tout son contenu, car la commande `cache:clear` ne fait pas que supprimer le cache, elle le reconstruit en partie, il restera donc des fichiers qu'on ne veut pas. Ensuite, envoyez tous vos fichiers et dossiers à la racine de votre hébergement, dans www/ sur OVH par exemple.

Que faire des vendors ?

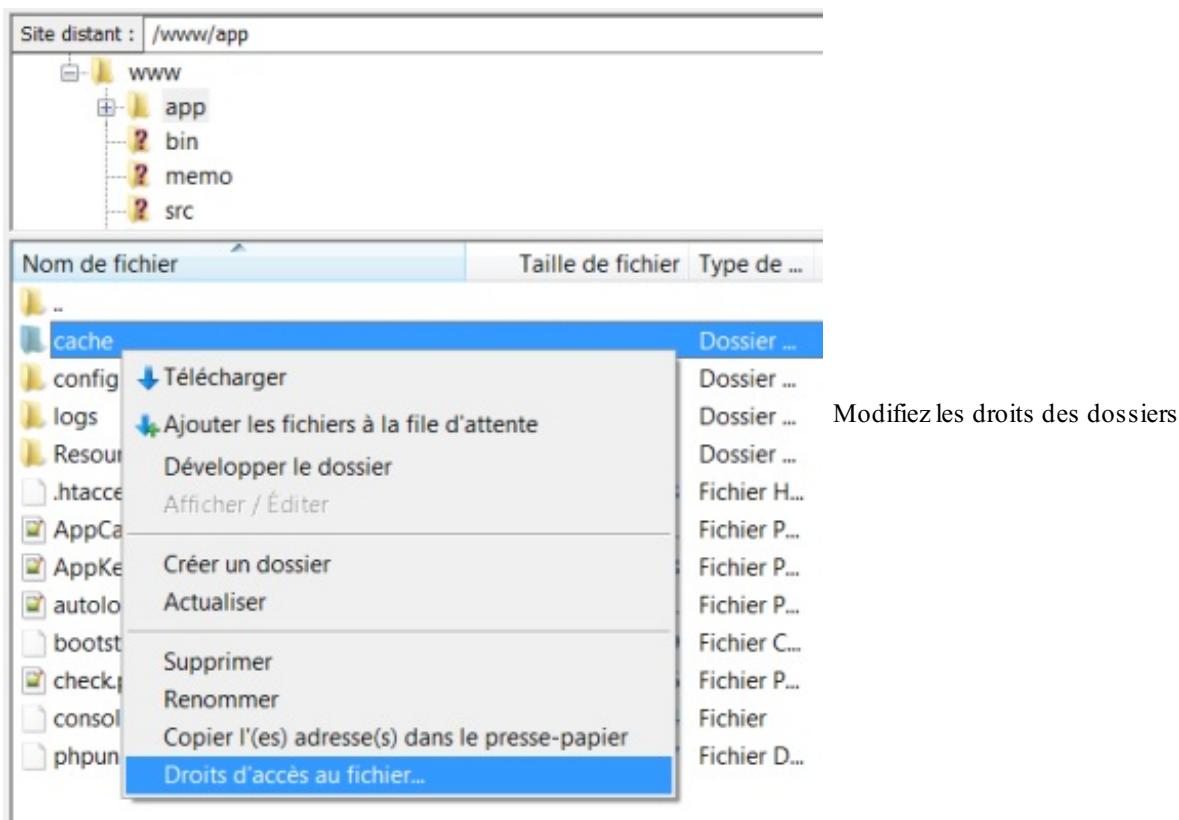
Si vous avez accès à Composer sur votre serveur, c'est le mieux. N'envoyez pas vos vendors à la main, ils sont assez lourds, mais envoyez bien les deux fichiers `composer.json` et `composer.lock`. Ensuite, sur votre serveur, exécutez la commande `php composer.phar install`. Je parle bien de la commande `install` et non `update`, qui va installer les **mêmes** versions des dépendances que vous avez en local. Cela se fait grâce au fichier `composer.lock` qui contient tous les numéros des versions installées justement.

Si vous n'avez pas accès à Composer sur votre serveur, alors contentez-vous d'envoyer le dossier `vendor` en même temps que le reste de votre application.

Régler les droits sur les dossiers app/cache et app/logs

Vous le savez maintenant, Symfony2 a besoin de pouvoir écrire dans deux répertoires : `app/cache` pour y mettre le cache de l'application et ainsi améliorer les performances, et `app/logs` pour y mettre l'historique des informations et erreurs rencontrées lors de l'exécution des pages.

Normalement, votre client FTP devrait vous permettre de régler les droits sur les dossiers. Avec FileZilla par exemple, un clic droit sur les dossiers `cache` et `logs` vous permet de définir les droits, comme à la figure suivante.



Assurez-vous d'accorder tous les droits (777) pour que Symfony2 puisse écrire à souhait dans ces dossiers.

S'autoriser l'environnement de développement

Pour exécuter les commandes Symfony2, notamment celles pour créer la base de données, il nous faut avoir accès à l'environnement de développement. Or, essayez d'accéder à votre `app_dev.php`... accès interdit ! En effet, si vous l'ouvrez, vous remarquez qu'il y a le même test sur l'IP qu'on avait rencontré dans `config.php`. Cette fois-ci, ne supprimez pas la condition, car vous aurez besoin d'accéder à l'environnement de développement dans le futur. Il faut donc que vous complétiez la condition avec votre adresse IP. Obtenez votre IP sur www.whatismyip.com, et rajoutez-la :

Code : PHP

```
<?php
// web/app_dev.php

// ...

if (!in_array(@$_SERVER['REMOTE_ADDR'], array(
    '127.0.0.1',
    '::1',
    '123.456.789.1'
))) {
    header('HTTP/1.0 403 Forbidden');
    exit('You are not allowed to access this file. Check
        '.basename(__FILE__).' for more information.');
}
```

Voilà, vous avez maintenant accès à l'environnement de développement et, surtout, à la console. 😊



Bien sûr, si vous avez une IP dynamique, il vous faudra la mettre à jour à chaque changement.

Mettre en place la base de données

Il ne manque pas grand-chose avant que votre site ne soit opérationnel. Il faut notamment s'attaquer à la base de données. Pour cela, modifiez le fichier `app/config/parameters.yml` de votre serveur afin d'adapter les valeurs des paramètres `database_*`.

Généralement sur un hébergement mutualisé vous n'avez pas le choix dans la base de données, et vous n'avez pas les droits pour en créer. Mais si ce n'est pas le cas, alors il faut créer la base de données que vous avez renseignée dans le fichier `parameters.yml`, en exécutant cette commande :

Code : Console

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

Puis, dans tous les cas, remplissez la base de données avec les tables correspondant à vos entités :

Code : Console

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

S'assurer que tout fonctionne

Ça y est, votre site devrait être opérationnel dès maintenant ! Vérifiez que tout fonctionne bien dans l'environnement de production.



En cas de page blanche ou d'erreur 500 pas très bavarde : soit vous allez voir les logs dans `app/logs/prod`, soit vous activez le mode `debugger` dans `app.php` comme on l'a fait précédemment. Dans tous les cas, pas de panique : si votre site fonctionnait très bien en local, l'erreur est souvent très bête sur le serveur (problème de casse, oubli, etc.).

Avoir de belles URL

Si votre site fonctionne bien, vous devez sûrement avoir ce genre d'URL pour l'instant : `www.votre-site.com/web/app.php`. On est d'accord, on ne va pas rester avec ces horribles URL !

Pour cela il faut utiliser l'[URL Rewriting](#), une fonctionnalité du serveur web Apache (rien à voir avec Symfony2). L'objectif est que les requêtes `/blog` et `/css/style.css` arrivent respectivement sur `/web/blog` et `/web/css/style.css`.

Méthode .htaccess

Pour faire cela avec un `.htaccess`, rajoutez donc ces lignes dans un `.htaccess` à la racine de votre serveur :

Code : Autre

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ web/$1 [QSA,L]
</IfModule>
```

C'est tout ! En effet, c'est déjà bon pour les fichiers CSS, mais pour l'URL `/blog` il faut qu'au final elle arrive sur `/web/app.php/blog`. En fait il y a déjà un `.htaccess` dans le répertoire `/web`. Ouvrez-le, il contient ce qu'il faut. Pour résumer, l'URL `/blog` va être réécrite en `/web/blog` par notre `.htaccess` à la racine, puis être à nouveau réécrite en `/web/app.php/blog` par le `.htaccess` de Symfony2 situé dans le répertoire `/web`.

Méthode VirtualHost

Si vous avez accès à la configuration du serveur HTTP Apache sur votre serveur, cette solution est à préférer. Vous pouvez l'essayer sur votre serveur local, où vous avez évidemment tous les droits.

Pour cela il faut créer un VirtualHost, c'est-à-dire un domaine virtuel sur lequel Apache va créer un raccourci. Autrement dit, au lieu d'accéder à `http://localhost/Symfony/web`, vous allez accéder à `http://symfony.local`. On va dire à Apache que le domaine `symfony.local` doit pointer directement vers le répertoire `Symfony/web` qui se trouve à la racine de votre serveur web.



Je le fais ici en local avec le domaine arbitraire `symfony.local`, mais si vous avez un vrai nom de domaine du genre `www.votreSite.com`, adaptez le code. 😊

Voici la configuration à rajouter dans le fichier de configuration d'Apache, `httpd.conf` :

Code : Autre

```
# Sous wamp : C:\wamp\bin\apache\apache2.2.22\conf\httpd.conf

<VirtualHost *:80>
    ServerName symfony.local
    DocumentRoot "C:/wamp/www/Symfony"

    <Directory "C:/wamp/www/Symfony">
        DirectoryIndex app.php
        Options -Indexes
        AllowOverride All
        Allow from All
    </Directory>
</VirtualHost>
```



Pour tester en local, il reste un petit détail : faire correspondre le domaine « `symfony.local` » à votre propre PC, soit `localhost` (ou encore `127.0.0.1`). Pour cela il faut modifier le fichier `hosts`, que vous trouverez ici :

`C:\Windows\System32\Drivers\etc\hosts`. Rajoutez simplement une ligne avec :
`127.0.0.1 symfony.local`. Ainsi, quand vous tapez `http://symfony.local` dans votre navigateur, Windows lui dira d'adresser sa requête à l'adresse IP `127.0.0.1`, c'est-à-dire votre propre PC, et c'est votre serveur Apache qui recevra la requête.

Et profitez !

Et voilà, votre site est pleinement opérationnel, profitez-en !



Et n'oubliez pas, à chaque modification de code source que vous envoyez sur le serveur, vous devez obligatoirement vider le cache de l'environnement de production ! L'environnement de production ne fonctionne pas comme l'environnement de développement : il ne vide jamais le cache tout seul, jamais !

Les outils pour déployer votre projet

Bien sûr, les méthodes que je vous ai données dans ce chapitre sont assez sommaires. Cela permet juste de vous expliquer quels sont les points particuliers du déploiement d'un projet Symfony2. Mais si votre projet est assez grand, vous devez penser à utiliser des outils adaptés pour le déployer sur votre serveur. Je vous invite notamment à jeter un œil à Capifony : capifony.org, un outil Ruby qui permet d'automatiser pas mal de choses que nous venons de voir. Je n'irai pas plus loin sur ce point, à vous d'investiguer !

En résumé

- Avant tout déploiement, préparez bien votre application en local.
- N'oubliez pas de personnaliser les pages d'erreur, d'installer une console via navigateur si vous êtes en hébergement

- mutualisé, et de vider le cache.
- Vérifiez la configuration de votre serveur, et adaptez-la si nécessaire.
 - Envoyez tous vos fichiers sur le serveur, et assurez-vous d'avoir de belles URL grâce aux `.htaccess` ou à un `VirtualHost`.
 - C'est tout bon !

Toutes les bonnes choses ont une fin !

Ce cours touche à sa fin. Mais il ne faut pas vous arrêter en si bon chemin !

Symfony2 est un framework avec des possibilités immenses, un nombre de bundles incalculable, bref, son apprentissage ne se termine jamais. Je vous invite donc à bien suivre tous les liens vers les pages des documentations officielles et des bundles que je vous ai donnés tout au long de ce cours. Cela vous permettra de perfectionner vos connaissances, et d'être au courant des fonctionnalités applicables à votre projet.

Le code complet du cours

Vous trouverez le code complet du blog créé à l'aide de ce cours à l'adresse suivante : www.tutoriel-symfony2.fr/livre/codesource.

N'hésitez pas à parcourir le code et à vérifier que le vôtre correspond bien. Ce blog peut également vous servir de blog tout fait, libre à vous de vous en servir et de vous en inspirer.

Plus de lecture sur mon blog

Vous pouvez également visiter mon blog sur Symfony2 : www.tutoriel-symfony2.fr, dans lequel je vous tiens au courant des améliorations et corrections du cours, ainsi que des trucs et astuces très pratiques sur Symfony2. C'est un vrai blog, basé sur le code construit grâce au cours, et j'espère vous donner quelques tuyaux intéressants !

Licences

La licence de ce cours est la [Creative Commons BY-NC-SA](#).

Certaines images de ce cours sont tirées de la documentation officielle. Elles sont donc soumises à la licence suivante :

Citation : Sensio Labs

Copyright (c) 2004-2010 Fabien Potencier

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.