Symfony

Tout le cours s’articule autour de la création d’une plateforme d’échange.

Table des matières

[Installer symfony 13](#_Toc469390952)

[Découvrir Symfony 13](#_Toc469390953)

[Les répertoires : 13](#_Toc469390954)

[Le contrôle frontal : 13](#_Toc469390955)

[Deux environnements de travail : 13](#_Toc469390956)

[Que contrôle le contrôle frontal ? 14](#_Toc469390957)

[L’architecture conceptuelle : 14](#_Toc469390958)

[Les Bundles : 15](#_Toc469390959)

[Structure d’un Bundle : 16](#_Toc469390960)

[Utiliser la console pour créer un Bundle : 16](#_Toc469390961)

[Créer le Bundle : 16](#_Toc469390962)

[Symfony enregistre le Bundle près du Kernel : 16](#_Toc469390963)

[Symfony enregistre nos routes auprès du Routeur : 17](#_Toc469390964)

[Premier script 18](#_Toc469390965)

[Créer la route : 18](#_Toc469390966)

[Créer le contrôleur : 19](#_Toc469390967)

[Les templates (ou vues) Twig : 20](#_Toc469390968)

[Appeler le template depuis le controller : 21](#_Toc469390969)

[Notre objectif : créer une plateforme d’annonce : 22](#_Toc469390970)

[Un peu de nettoyage : 22](#_Toc469390971)

[Schéma de développement sous Symfony : 23](#_Toc469390972)

[Le routeur de Symfony 24](#_Toc469390973)

[Le fonctionnement : 24](#_Toc469390974)

[Fonctionnement du routeur : 24](#_Toc469390975)

[Les routes de base : 25](#_Toc469390976)

[Créer une route avec des paramètres : 26](#_Toc469390977)

[Les routes avancées : 27](#_Toc469390978)

[Paramètres facultatifs : 27](#_Toc469390979)

[Utiliser des paramètres systèmes : 28](#_Toc469390980)

[Ajouter un préfixe lors de l’import de nos routes : 28](#_Toc469390981)

[Générer des URL : 28](#_Toc469390982)

[Pourquoi générer des URL ? 28](#_Toc469390983)

[Comment générer des URL : 29](#_Toc469390984)

[Application : les routes de notre plateforme : 30](#_Toc469390985)

[Les contrôleurs avec Symfony 31](#_Toc469390986)

[Le rôle du contrôleur : 31](#_Toc469390987)

[L’objet Request : 32](#_Toc469390988)

[Les paramètres contenus dans les routes : 32](#_Toc469390989)

[Les paramètres hors routes : 32](#_Toc469390990)

[Les autres méthodes de l’objet Request : 34](#_Toc469390991)

[Réponses et vues : 35](#_Toc469390992)

[Réponse et redirection : 36](#_Toc469390993)

[Changer le Content-type de la réponse : 37](#_Toc469390994)

[Manipuler la session : 38](#_Toc469390995)

[Le contrôleur de note plateforme : 40](#_Toc469390996)

[Le moteur de template Twig 42](#_Toc469390997)

[Les templates Twig : 42](#_Toc469390998)

[Intérêt : 42](#_Toc469390999)

[Page web, email … 42](#_Toc469391000)

[En pratique : 42](#_Toc469391001)

[Afficher des variables : 43](#_Toc469391002)

[Twig et la sécurité : 44](#_Toc469391003)

[Les variables globales : 45](#_Toc469391004)

[Structure et contrôle des expressions : 46](#_Toc469391005)

[Les structures de contrôle : 46](#_Toc469391006)

[Hériter et inclure des templates : 48](#_Toc469391007)

[La pratique : 48](#_Toc469391008)

[Le nom du template père : 49](#_Toc469391009)

[Organiser son code en triple héritage : 49](#_Toc469391010)

[Héritage ou inclusion ? 50](#_Toc469391011)

[La pratique : 51](#_Toc469391012)

[Inclusion de contrôleur : 51](#_Toc469391013)

[Application : 52](#_Toc469391014)

[Layout général : 53](#_Toc469391015)

[Layout du Bundle : 54](#_Toc469391016)

[Templates finaux : 54](#_Toc469391017)

[Installer un Bundle avec Composer 55](#_Toc469391018)

[C’est quoi ? 55](#_Toc469391019)

[L’installation : 55](#_Toc469391020)

[Installer un Bundle : 56](#_Toc469391021)

[Installer un Bundle : 57](#_Toc469391022)

[Gérer l’autoload : 58](#_Toc469391023)

[Services, théories et créations 59](#_Toc469391024)

[Pourquoi utiliser des services ? 59](#_Toc469391025)

[Qu’est-ce qu’un service ? 59](#_Toc469391026)

[Avantage de la POS (Programmation Orientée Service) : 59](#_Toc469391027)

[Le conteneur des services : 60](#_Toc469391028)

[Comment définir une dépendance entre service ? 61](#_Toc469391029)

[Le partage des services : 62](#_Toc469391030)

[Utiliser un service en pratique : 62](#_Toc469391031)

[Récupérer un service : 62](#_Toc469391032)

[Créer un service simple : 63](#_Toc469391033)

[Création de la configuration du service : 63](#_Toc469391034)

[Utilisation des services : 64](#_Toc469391035)

[Créer un service avec des arguments : 65](#_Toc469391036)

[La couche métier : les entités : 66](#_Toc469391037)

[Notion d’ORM : 66](#_Toc469391038)

[Créer une première entité avec doctrine : 66](#_Toc469391039)

[Les commentaires : 67](#_Toc469391040)

[Générer une entité avec le générateur : 68](#_Toc469391041)

[Affiner l’entité avec de la logique métier : 69](#_Toc469391042)

[Tout sur le mapping : 70](#_Toc469391043)

[Entity : 70](#_Toc469391044)

[Class : 70](#_Toc469391045)

[Column : 70](#_Toc469391046)

[Manipuler ses entités avec doctrine : 72](#_Toc469391047)

[Matérialiser les tables en BDD : 72](#_Toc469391048)

[Modifier une entité : 73](#_Toc469391049)

[Enregistrer ses entités avec l’Entity Manager : 74](#_Toc469391050)

[Les services de Doctrine2 : 74](#_Toc469391051)

[EntityManager : 74](#_Toc469391052)

[Les repositories : 74](#_Toc469391053)

[Enregistrer dans la base : 75](#_Toc469391054)

[Doctrine utilise les transactions : 77](#_Toc469391055)

[Doctrin simplifie la vie : 77](#_Toc469391056)

[Les autres méthodes de l’EntityManager : 78](#_Toc469391057)

[Récupérer ses entités avec un repositories : 79](#_Toc469391058)

[Relation entre entité et Doctrine 80](#_Toc469391059)

[Présentation : 80](#_Toc469391060)

[Notion de propriétaire et d’inverse : 80](#_Toc469391061)

[Notion d’unidirectionnalité et bidirectionnalité : 81](#_Toc469391062)

[Relation OneToOne : 81](#_Toc469391063)

[Définition de la relation : 82](#_Toc469391064)

[Rendre une relation non-facultative : 83](#_Toc469391065)

[Les opérations de cascade : 83](#_Toc469391066)

[Relation ManyToOne : 85](#_Toc469391067)

[Définir la relation : 87](#_Toc469391068)

[Relation ManyToMany : 88](#_Toc469391069)

[Remplir la BDD avec les fixtures : 90](#_Toc469391070)

[Relation ManyToMany avec attribut : 93](#_Toc469391071)

[Les relations bidirectionnelles : 97](#_Toc469391072)

[Récupérer ses entités avec Doctrine 100](#_Toc469391073)

[Le rôle des repositories : 100](#_Toc469391074)

[Deux façons pour contruire les query et récupérer des entités : 100](#_Toc469391075)

[Les méthodes de récupérations de base : 101](#_Toc469391076)

[Find($id) : 101](#_Toc469391077)

[findAll() : 101](#_Toc469391078)

[findBy() : 101](#_Toc469391079)

[findOneBy() : 102](#_Toc469391080)

[Les méthodes magiques : findByX et findOneByX : 102](#_Toc469391081)

[Méthodes de récupération personnelle : 103](#_Toc469391082)

[Le QueryBuilder : 103](#_Toc469391083)

[Méthode La Query : 106](#_Toc469391084)

[GetResult() : 106](#_Toc469391085)

[GetArrayResult() : 107](#_Toc469391086)

[GetOneOrNullResult() : 107](#_Toc469391087)

[getSingleResult() : 107](#_Toc469391088)

[Execute() : 107](#_Toc469391089)

[Utilisation du DQL : 108](#_Toc469391090)

[Utiliser les jointures dans les requêtes : 109](#_Toc469391091)

[Pourquoi utiliser les jointures ? 109](#_Toc469391092)

[Jointures avec le Query Builder : 109](#_Toc469391093)

[Evènements et extensions Doctrine 111](#_Toc469391094)

[Les évènements : 111](#_Toc469391095)

[Utiliser des services : 115](#_Toc469391096)

[Les extensions Doctrine : 117](#_Toc469391097)

[Installer le SoftDoctrineExtensionBundle : 117](#_Toc469391098)

[Créer des formulaires 118](#_Toc469391099)

[Gestion des formulaires : 118](#_Toc469391100)

[Qu’est ce que c’est ? 118](#_Toc469391101)

[Ajouter des champs : 120](#_Toc469391102)

[Gérer les valeurs par défaut : 122](#_Toc469391103)

[Personnaliser l’affichage du formulaire : 123](#_Toc469391104)

[Externaliser la définition de ses formulaires : 125](#_Toc469391105)

[Les formulaires imbriqués : 126](#_Toc469391106)

[Relation simple, imbriquer un seul formulaire : 127](#_Toc469391107)

[Relations multiples, imbriquer un ou plusieurs formualaires : 128](#_Toc469391108)

[Le type de champ Entity : 132](#_Toc469391109)

[L’option queryBuilder : 133](#_Toc469391110)

[Aller plus loin avec les forumaires : 134](#_Toc469391111)

[Héritage : 134](#_Toc469391112)

[Formulaire différent selon des paramètres : 135](#_Toc469391113)

[Type de champ File pour envoyer des fichiers : 137](#_Toc469391114)

[Type de champ File : 137](#_Toc469391115)

[Manipuler le fichier envoyé : 138](#_Toc469391116)

[Automatisation avec les évènements : 140](#_Toc469391117)

[Implémenter les méthodes des évènements : 140](#_Toc469391118)

[Valider ses données 144](#_Toc469391119)

[Pourquoi valider ses données ? 144](#_Toc469391120)

[Définir les règles de validations : 144](#_Toc469391121)

[Annotation : 144](#_Toc469391122)

[Les contraintes existantes : attention sensible à la casse ! 146](#_Toc469391123)

[Déclencher la validation : 149](#_Toc469391124)

[Le service Validator : 149](#_Toc469391125)

[Validation automatique sur les forumaires : 150](#_Toc469391126)

[Encore plus de règle de validation : 150](#_Toc469391127)

[Valider depuis un Getter : 150](#_Toc469391128)

[Valider intelligemment un attribut objet : 151](#_Toc469391129)

[Valider depuis un callback() : 152](#_Toc469391130)

[Valider un champ unique : 152](#_Toc469391131)

[Valider selon nos propres contraintes : 153](#_Toc469391132)

[Gérer la contrainte : 153](#_Toc469391133)

[Le validateur : 154](#_Toc469391134)

[Définition du service : 155](#_Toc469391135)

[Modification des contraintes : 156](#_Toc469391136)

[Modification du validateur : 156](#_Toc469391137)

[Securité et gestion des utilisateurs 157](#_Toc469391138)

[Authentification et autorisation : 157](#_Toc469391139)

[L’authentification : 157](#_Toc469391140)

[L’autorisation : 157](#_Toc469391141)

[Première approche de la sécurité : 158](#_Toc469391142)

[Fichier de configuration de la sécurité : 158](#_Toc469391143)

[Mettre en place un Pare-feu : 159](#_Toc469391144)

[Créer le Bundle OCUserBundle : 160](#_Toc469391145)

[Créer le formulaire de connexion : 161](#_Toc469391146)

[Les erreurs courantes : 162](#_Toc469391147)

[Récupérer l’utilisateur courant : 163](#_Toc469391148)

[Depuis le contrôleur ou un service : 163](#_Toc469391149)

[Depuis une vue : 164](#_Toc469391150)

[Gestion des autorisations avec les rôles : 164](#_Toc469391151)

[Définition des rôles : 164](#_Toc469391152)

[Le rôle de l’utilisateur : 165](#_Toc469391153)

[Utiliser des utilisateurs de la BDD : 167](#_Toc469391154)

[Créer la classe utilisateur : 167](#_Toc469391155)

[Des utilisateurs de test : 168](#_Toc469391156)

[Définir l’encoder pour une nouvelle classe utilisateur : 169](#_Toc469391157)

[Définir le fournisseur utilisateur : 169](#_Toc469391158)

[Créer notre fournisseur entity : 170](#_Toc469391159)

[Dire au pare-feu d’utiliser le nouveau provider : 170](#_Toc469391160)

[Manipuler vos utilisateurs : 170](#_Toc469391161)

[Utiliser FOSUserBundle : 171](#_Toc469391162)

[Hériter FOSUserBundle depuis notre OCUserBundle : 171](#_Toc469391163)

[Modifier notre entité User : 172](#_Toc469391164)

[Configurer le Bundle : 173](#_Toc469391165)

[Configuration de la sécurité pour utiliser le Bundle : 173](#_Toc469391166)

[Configuration de FOSUserBundle : 174](#_Toc469391167)

[Personnalisation esthétique du Bundle : 176](#_Toc469391168)

[Traduire les messages : 176](#_Toc469391169)

[Afficher une barre utilisateur : 177](#_Toc469391170)

[Manipuler les utilisateurs avec FOSUserBundle : 177](#_Toc469391171)

[Les services utilisation poussée 178](#_Toc469391172)

[Les tags sur les serices : 178](#_Toc469391173)

[Comprendre les tags à travers Twig : 178](#_Toc469391174)

[Appliquer un tag à un service : 178](#_Toc469391175)

[Une classe qui implémente une interface : 179](#_Toc469391176)

[Hérite le code qui sera exécuté : 180](#_Toc469391177)

[Méthodologie : 181](#_Toc469391178)

[Les principaux tags : 181](#_Toc469391179)

[Les types de champs du formulaire : 181](#_Toc469391180)

[Dépendances optionnelles : les appels de méthode : 183](#_Toc469391181)

[Les dépendances optionnelles : 183](#_Toc469391182)

[Les calls : 183](#_Toc469391183)

[Les services courants de Symfony : 184](#_Toc469391184)

[Le gestionnaire d’évènement de Symfony 187](#_Toc469391185)

[Des évènements, pour quoi faire ? 187](#_Toc469391186)

[Qu’est ce qu’un gestionnaire d’évènement ? 187](#_Toc469391187)

[Ecouter des évènements : 188](#_Toc469391188)

[Créer un service et son listener : 188](#_Toc469391189)

[Ecouter un évènement : 189](#_Toc469391190)

[Création de la méthode à exécuter avec le Listener : 191](#_Toc469391191)

[Les évènements Symfony et les nôtres : 193](#_Toc469391192)

[Les évènements Symfony : 193](#_Toc469391193)

[Créer nos propres évènements : 197](#_Toc469391194)

[Définir la liste de nos évènements : 198](#_Toc469391195)

[Construire la classe de l’évènement : 198](#_Toc469391196)

[Déclencher l’évènement : 199](#_Toc469391197)

[Ecouter l’évènement : 200](#_Toc469391198)

[Aller plus loin : 201](#_Toc469391199)

[Les souscripteurs d’évènements : 201](#_Toc469391200)

[La priorité des évènements : 203](#_Toc469391201)

[La propagation d’évènement : 203](#_Toc469391202)

[Traduire son site 204](#_Toc469391203)

[Le principe : 204](#_Toc469391204)

[Configuration : 205](#_Toc469391205)

[Bonjour tout le monde : 206](#_Toc469391206)

[Dire à Symfony de traduire : 206](#_Toc469391207)

[Le filtre Twig : 206](#_Toc469391208)

[La balise de bloc Twig : 206](#_Toc469391209)

[Le service translator : 207](#_Toc469391210)

[Le catalogue : 207](#_Toc469391211)

[Le format XLIFF 207](#_Toc469391212)

[Format YAML : 208](#_Toc469391213)

[Le format PHP : 208](#_Toc469391214)

[Mise en cache du catalogue : 208](#_Toc469391215)

[Notre traduction : 208](#_Toc469391216)

[Ajouter un nouveau message à traduire : 209](#_Toc469391217)

[Extraire les chaînes d’un site existant : 209](#_Toc469391218)

[Traduire une nouvelle ligne : 209](#_Toc469391219)

[Récupérer la locale de l’utilisateur : 210](#_Toc469391220)

[Routing et locale : 210](#_Toc469391221)

[Organiser vos catalogues : 211](#_Toc469391222)

[Nicher les traductions : 212](#_Toc469391223)

[Retoure à la ligne : 212](#_Toc469391224)

[Utiliser des listes : 213](#_Toc469391225)

[Utiliser les domaines : 213](#_Toc469391226)

[Domaines et Bundles : 214](#_Toc469391227)

[Un domaine special : validator : 214](#_Toc469391228)

[Traduction dépendante de variables : 214](#_Toc469391229)

[Les placeholders : 214](#_Toc469391230)

[Les placeholders dans le domaine validator : 215](#_Toc469391231)

[Gestion des pluriels : 215](#_Toc469391232)

[Afficher les dates au format local : 216](#_Toc469391233)

[Récapitulatif : 217](#_Toc469391234)

[Convertir les paramètres de requête 219](#_Toc469391235)

[Pourquoi un paramConverteur ? 219](#_Toc469391236)

[Récupérer des entités doctrine avant le contrôleur : 219](#_Toc469391237)

[Un paramConverteur utile : DoctrineParamConverter : 219](#_Toc469391238)

[Utilisation des paramConverters existants 220](#_Toc469391239)

[Utiliser le paramConverter Doctrine : 220](#_Toc469391240)

[S’appuyer sur l’ID et le typage de l’agument : 220](#_Toc469391241)

[Utiliser l’annotation pour faire correspondre la route et l’identité : 220](#_Toc469391242)

[Utiliser les annotations sur plusieurs arguments : 221](#_Toc469391243)

[Utiliser le paramConverter Datetime : 222](#_Toc469391244)

[Créer ses propres paramConverters : 222](#_Toc469391245)

[Comment sont exécutés les paramConverters ? 222](#_Toc469391246)

[Comment Symfony trouve les convertisseurs ? 223](#_Toc469391247)

[Créer un convertisseur : 223](#_Toc469391248)

[Support : 224](#_Toc469391249)

[Apply : 224](#_Toc469391250)

[L’exemple de notre JsonparaConverter : 224](#_Toc469391251)

[Personnaliser les pages d’erreurs 226](#_Toc469391252)

[Constater les pages d’erreur : 226](#_Toc469391253)

[Localiser les vues concernées : 226](#_Toc469391254)

[Remplacer les vues d’un Bundle : 226](#_Toc469391255)

[Comportement de Twig : 227](#_Toc469391256)

[Pourquoi toutes les erreurs error.XXX.twig sont dans le répertoire Exception ? 227](#_Toc469391257)

[Remplacer les templates exceptions du twig : 227](#_Toc469391258)

[Créer la nouvelle vue : 227](#_Toc469391259)

[Utiliser Assetic pour gérer le CSS et JS 229](#_Toc469391260)

[Entre vitesse et lisibilité 229](#_Toc469391261)

[Comment optimiser le front-end ? 229](#_Toc469391262)

[On peut aussi améliorer le temps de chargement : 229](#_Toc469391263)

[La pratique : 229](#_Toc469391264)

[Installer Assetic : 230](#_Toc469391265)

[Servir des ressources : 230](#_Toc469391266)

[Le filtre csswrite : 231](#_Toc469391267)

[Les ciltres scssphp et jsqueeze : 232](#_Toc469391268)

[Appliquer les chagements au mode prod’ : 232](#_Toc469391269)

[Exporter les fichiers CSS et JS : 233](#_Toc469391270)

[Utiliser la console directement depuis le navigateur 234](#_Toc469391271)

[Le composant console de Symfony : 234](#_Toc469391272)

[Les commandes sont en PHP : 234](#_Toc469391273)

[Exemple d’une commande : 234](#_Toc469391274)

[Exemple de commande : 235](#_Toc469391275)

[Utiliser un console Bundle : 236](#_Toc469391276)

[Installer CoreSphereConsoleBundle : 237](#_Toc469391277)

[Enregistrement des routes : 237](#_Toc469391278)

[Publier les Assets : 238](#_Toc469391279)

[Utilisation de la console : 238](#_Toc469391280)

[Déployer son site en production 238](#_Toc469391281)

[Préparer l’appli en local : 238](#_Toc469391282)

[Vérifier la qualité de son code : 239](#_Toc469391283)

[Vérifier la sécurité des dépendances : 239](#_Toc469391284)

[Vérifier et préparer le serveur de production : 239](#_Toc469391285)

[Vérifier la compatibilité : 240](#_Toc469391286)

[Déployer l’application : 241](#_Toc469391287)

[Méthode 1 : envoyer par FTP : 241](#_Toc469391288)

[Méthode 2 : utiliser Capifony : 241](#_Toc469391289)

[Les derniers préparatifs une fois le site en ligne : 242](#_Toc469391290)

[Mettre en place la BDD : 242](#_Toc469391291)

[Avoir de belles URL : 242](#_Toc469391292)

[Mise à jour de la BDD : 243](#_Toc469391293)

# Installer symfony

On doit télécharger un certificat de sécurité : présent dans le dossier symfony.

Dans php.ini on doit spécifier dans CURL =, le chemin d’accès absolu vers ce fichier. Ensuite on peut enfin créer un fichier de travail avec « php symfony.phar new test ».

Sous Linux, une partie sur les droits d’accès est à revoir sur le cours.

# Découvrir Symfony

## Les répertoires :

* App : contient tout ce qui concerne le site sauf son code source.
* Bin : contient les exécutables dont nous avons besoin, comme des commandes PHP.
* Tests : contient tous les tests de l’application, pas évoqué ici mais utile.
* Var : tout ce qui est utile au bon fonctionnement. On n’écrit pas dedans nous-même.
* Vendor : contient toutes les bibliothèques externes au projet.
* Web : contient tout ce qui est destiné au visiteur : image, css, JS … C’est aussi le seul qui sera accessible par celui-ci.

On utilisera donc toujours Symfony/web, pour être sûr que le visiteur n’accède qu’à celui-ci. On passe donc du temps dans le répertoire **SRC** pour travailler sur les **BUNDLES**. Au répertoire **APP** pour configurer les applications. Et **VENDOR** pour installer des bundles téléchargés.

### Le contrôle frontal :

C’est l’index.php, la page d’entrée du site. Avec Symfony on en a deux :

* Un pour les visiteurs : app.
* Un pour le dév’ app\_dev.

### Deux environnements de travail :

Chaque personne n’a pas besoin des mêmes informations sur la page :

* 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.

Si on essaye d’accéder à la page app.php/\_profiler au lieu de app\_dev, on n’a rien, parce qu’on n’a pas défini pour l’url \_profiler. On obtient une error 404. Quand on essaye d’aller sur une page qui n’existe pas on obtient une tout autre page.

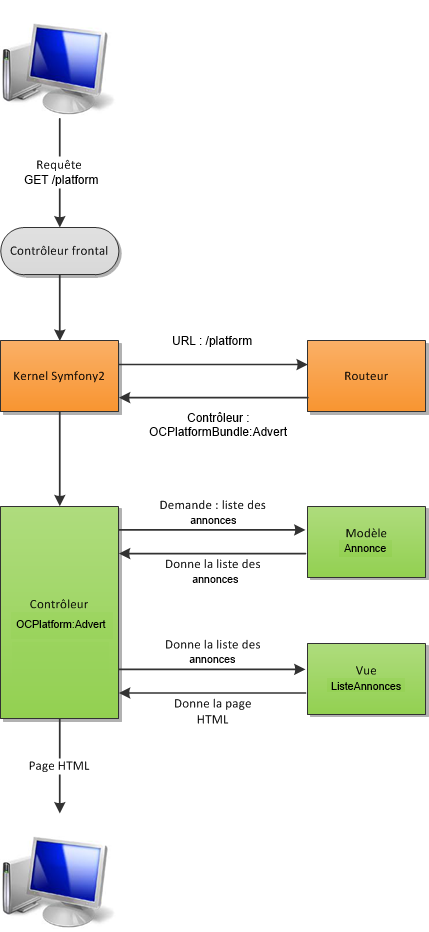
Pour trouver les erreurs on regarde le fichier var/logs/prod.log.

### Que contrôle le contrôle frontal ?

Ici on se limite à appeler le noyau (kernel) de Symfony.

## L’architecture conceptuelle :

Le modèle MVC, on peut voir le chemin d’une requête :



* **Le contrôleur** : son rôle est de générer la réponse à la requête http demandée. Il analyse et traite la réponse de l’utilisateur.
* **Le modèle** : gère les données et le contenu.
* **La vue** : affiche les pages.

L’utilisateur fait une requête 🡺 le contrôleur les reçoit et demande au modèle 🡺 il les renvoie au contrôleur 🡺 qui envoie à la vue et l’affiche 🡺 le contrôleur a son information.

## Les Bundles :

On regroupe les fonctionnalités par Bundle, chaque Bundle gère une tâche bien précise.

Exemple : le Bundle blog contiendra contrôleur, modèle, vue, code JS et CSS qui concerne uniquement le blog.

* 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 dans un site e-commerce par exemple.

On peut donc importer des exporter des fonctionnalités, des Bundles que l’on réutilisera par la suite.

Quand on fait son programme, pour être rapide si on ne souhaite pas le partager on fait souvent dans un seul et même Bundle appelé souvent Core ou App.  
Les Bundles de la communauté sur ce site : [les Bundles](http://knpbundles.com).

**Quelques Bundles utiles :**

* [FOSUserBundle](http://knpbundles.com/FriendsOfSymfony/FOSUserBundle) pour gérer ses utilisateurs.
* [FOSCommentBundle](http://knpbundles.com/FriendsOfSymfony/FOSCommentBundle) gère les commentaires.
* [GravatarBundle](http://knpbundles.com/ornicar/GravatarBundle) gère les avatars des utilisateurs depuis Gravatar.

### Structure d’un Bundle :

/Controller          | Contient vos contrôleurs

/DependencyInjection | Contient des informations sur votre bundle (chargement automatique de la configuration par exemple)

/Entity              | Contient vos modèles

/Form                | Contient vos éventuels formulaires

/Resources

-- /config             | Contient les fichiers de configuration de votre bundle (nous placerons les routes ici, par exemple)

-- /public             | Contient les fichiers publics de votre bundle : fichiers CSS et JavaScript, images, etc.

-- /views              | Contient les vues de notre bundle, les templates Twig

# Utiliser la console pour créer un Bundle :

Symfony utilise des commandes disponibles sur la console et non le navigateur.

Pour faire une commande, on se place dans le dossier de Symfony et on exécute une commande PHP : php bin/console.

Grace à ça on pourra par exemple, créer une BDD ou en vider une sans phpmyadmin, vider le cache etc …

Ici on va générer le code de base pour les Bundles qui est identiques à chacun. Le code bin/console fait la même chose que app.php mais définit la requête comme venant de la console.

## Créer le Bundle :

On utilise toujours la même syntaxe :

Php bin/console generate :bundle

Pour nommer son Bundle comme ainsi : OC/PlatformBundle, on a OC qui est la présentation de site, entreprise … en racine, puis le nom du Bundle puis « Bundle » qui est obligatoire. Le nom final est sans slash : OCPlatformBundle.

Ensuite on a le choix du format (plus tard) ici yml.

### Symfony enregistre le Bundle près du Kernel :

La configuration se fait dans App, ainsi on a app/AppKernel.php dans lequel il faut indiquer que l’on veut charger notre Bundle.

Les premières lignes indiquent les Bundles à charger pour la production et les suivants (dans le if) pour le développement. Le générateur de Bundle a généré lui-même une ligne supplémentaire en production avec notre nouveau Bundle.

### Symfony enregistre nos routes auprès du Routeur :

Routeur : défini quel contrôleur appliqué en fonction de l’URL appelé. Pour cela il utilise les routes.

Chaque Bundle a ses propres routes, que l’on peut voir ici : src/OC/PlatformBundle/Resources/config/routing.yml.

Pour le moment il n’y en a que une :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_homepage:

path: /

defaults: { \_controller: OCPlatformBundle:Default:index }

Les routes ne se chargent pas toute seul, il faut dire au routeur de les charger. Ca se fait dans « application »

# app/config/routing.yml

oc\_platform:

resource: "@OCPlatformBundle/Resources/config/routing.yml"

prefix: /

# ...

**Ce qu’il faut 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 OCPlatformBundle.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 tâches se font automatiquement avec le générateur. Mais elles sont à faire manuellement si on ne l’utilise pas.

# Premier script

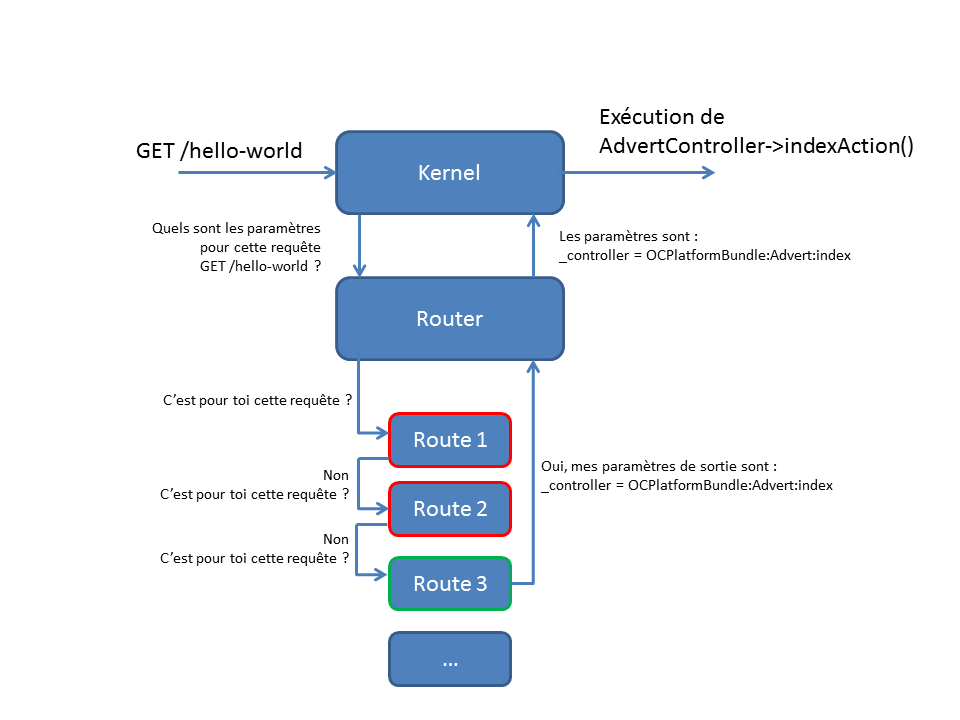
## Créer la route :

On va dans src/oc/PlateformBundle.

Pour créer une page il faut définir l’URL à laquelle elle est disponible. Il faut donc créer le route de cette page.

**L’objectif du routeur est de faire correspondance entre l’URL et des paramètres.**

Exemple : on pourrait avoir une route qui dit : quand l’URL est /hello-world, le contrôleur à exécuter est « Advert ».



Donc dans notre Bundle OC/PlateformBundle on exécute le contrôleur Advert.

# src/OC/PlatformBundle/Resources/config/routing.yml

hello\_the\_world:

path: /hello-world

defaults: { \_controller: OCPlatformBundle:Advert:index }

Attention l'indentation se fait avec 4 espaces par niveau, et non 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 YAML (.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.

* Hello\_the\_world est le nom de la route. Il doit être **unique**.
* Path est l’URL à laquelle on souhaite qu’il soit accessible.
* Defaults correspond aux paramètres de la route.
  + \_controller correspond à l’action que l’on veut exécuter et au contrôleur que l’on va appeler. Action = index, contrôleur = Advert.

On donne la valeur OCPlateformBundle:Advert:index à \_controller.

* OCPlatformBundle est le nom du Bundle, celui dans lequel Symfony ira chercher le contrôleur.
* Advert est le nom du contrôleur à ouvrir : src/OC/PlatformBundle/Controller/AdvertController.php.
* Index est le nom de la méthode à exécuter au sein du contrôleur.

On a déjà vu dans app/config/routing.yml que le fichier route du Bundle est déjà inclus dans la configuration générale.

On aurait pu ajouter hello\_the\_world directement dans le routing.yml mais on irait à l’encontre des principes des Bundles.

Au moins plus tard on pourra modifier le fichier : **src/OC/PlatformBundle/Resources/config/routing.yml** au lieu de **app/config/routing.yml**.

## Créer le contrôleur :

Dans le Bundle, les contrôleurs se trouvent dans « Controller ». On a comme convention d’avoir le suffixe « controller » dans les fichiers :

Src/OC/PlatformBudle/Controller/AdvertController.php.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController

{

public function indexAction()

{

return new *Response*("Notre propre Hello World !");

}

}

* Ligne 5 : on se place dans le namespace des contrôleurs de notre bundle. Suivez juste la structure des répertoires dans lequel se trouve le contrôleur.
* Ligne 7 : notre contrôleur va utiliser l'objet Response, il faut donc le définir grâce au use.
* Ligne 9 : le nom de notre contrôleur respecte le nom du fichier pour que l'*autoload* fonctionne.
* Ligne 11 : on définit la méthode indexAction(). N'oubliez pas de mettre le suffixe Action derrière le nom de la méthode.
* Ligne 13 : on crée une réponse toute simple. L'argument de l'objet Response est le contenu de la page que vous envoyez au visiteur, ici « Notre propre Hello World ! ». Puis on retourne cet objet.

Si on parle de l’action « index », le contrôleur doit définir la méthode indexAction().

## Les templates (ou vues) Twig :

**Twig est un moteur de template** : un script qui affiche le contenu dynamiquement de note page HTML. Chaque moteur de template a son propre langage.

Exemple : code 1 en PHP, code 2 en langage Twig

<!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->getHref(); ?>"><?php echo $item->getTitre(); ?></a>

</li>

<?php } ?>

</ul>

</body>

</html>

<!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>

On va donc utiliser le moteur Twig au lieu d’afficher notre texte de manière standard.

Le répertoire des templates est dans Resources/views.

On va donc y insérer :

{# src/OC/PlatformBundle/Resources/views/Advert/index.html.twig #}

<!DOCTYPE html>

<html>

<head>

<title>Bienvenue sur ma première page avec OpenClassrooms !</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>

### Appeler le template depuis le controller :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

// N'oubliez pas ce use :

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function indexAction()

{

$content = $this->get('templating')->render('OCPlatformBundle:Advert:index.html.twig');

return new *Response*($content);

}

}

Dans la ligne de $content on a : NomDuBundle :NomDuContrôleur :NomDeLAction.

$this->get(‘templating’) retourne un objet dont le nom ici est templating. La méthode render de l’objet templating permet de récupérer le contenu.

**La toolbar en bas de page apparait quand on a la balise fermante de Body.**

On peut modifier le contenu en ajoutant un tableau :

<?php

$content = $this

->get('templating')

->render('OCPlatformBundle:Advert:index.html.twig', array('nom' => 'winzou'))

;

## Notre objectif : créer une plateforme d’annonce :

La plateforme que nous allons créer est très simple. En voici les grandes lignes :

* Nous aurons des annonces (*advert* en anglais) de mission : développement d'un site internet, création d'une maquette, intégration HTML, etc. ;
* Nous pourrons consulter, créer, modifier et rechercher des annonces ;
* À chaque annonce, nous pourrons lier une image d'illustration ;
* À chaque annonce, nous pourrons lier plusieurs candidatures (*application* en anglais) ;
* Nous aurons plusieurs catégories (Développement, Graphisme, etc.) qui seront liées aux annonces. Nous pourrons créer, modifier et supprimer ces catégories ;
* À chaque annonce, nous pourrons enfin lier des niveaux de compétence requis (Expert en PHP, maîtrise de Photoshop, etc.).
* Au début, nous n'aurons pas de système de gestion des utilisateurs : nous devrons saisir notre nom lorsque nous rédigerons une annonce. Puis, nous rajouterons la couche utilisateur.

Pour éviter les redondances, on supprime :

* Controller/DefaultController.php
* Resources/views/Default
* Oc\_platform\_homepage dans Resources/config/routing.yml

### Un peu de nettoyage :

On va supprimer tous les fichiers « par défaut » de l’application :

* Le contrôleur Controller/DefaultController.php ;
* Le répertoire de vues Resources/views/Default ;
* La route oc\_platform\_homepage dans Resources/config/routing.yml.

Supprimez également tout ce qui concerne le bundle AppBundle,  un bundle de démonstration intégré dans la distribution standard de Symfony2 et dont nous ne nous servirons pas :

* Le répertoire src/AppBundle  ;
* La lignes 19 du fichier app/AppKernel.php, celle qui active le bundle : new AppBundle\AppBundle() ;
* Les lignes 7 à 9 du fichier app/config/routing.yml, celles qui importent le fichier de route du bundle AppBundle (app: resource: "@AppBundle/Controller/" type: annotation" ).

Ainsi on va se concentrer uniquement sur nos Bundles.

### Schéma de développement sous Symfony :

Il faut penser à vider le cache !

* En mode prod’, inutile, on peut vider la cache à la main avec la commande : php bin/console cache :clear –env=prod.
* En mode dév’, Symfony construit une bonne partie du cache à la prochaine page que l’on charge, donc pas besoin de le vider. Au cas où : php bin/console cache :clear.

Parfois ça ne marche qu’au deuxième coup. Sinon pour le vider à la main on supprime les répertoires var/cache/dev et var/cache/prod.

On respecte donc ce schéma :

* 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 ;

**En résumé :**

On crée un Bundle, dans lequel on a le modèle MVC ainsi que du code autre. Le bundle est stocké dans **SRC**.

Php bin/console generate :bundle

Dedans on a :

* Controller dans lequel on a le nom du controller et à l’intérieur de celui-ci des méthodes.
* Resources dans lequel on a :
  + Configu pour définir les routes avec le Routeur.
  + Views pour les templates (ici Twig).

On définit les routes dans notre Bundle pour éviter de le faire dans App (qui marcherait aussi). Par contre il faut définir dans route.yml de App que notre Bundle possède des routes :

oc\_platform:

resource: "@OCPlatformBundle/Resources/config/routing.yml"

prefix: /

# Le routeur de Symfony

Comme vu précédemment, le routeur a pour rôle **de définir quel contrôleur appelé en fonction de l’URL**.

## Le fonctionnement :

Le paramètre qui nous intéresse le plus est *\_controller* qui correspond au contrôleur à exécuter.

On va maintenant travailler avec cette feuile de route :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_home:

path: /platform

defaults: { \_controller: OCPlatformBundle:Advert:index }

oc\_platform\_view:

path: /platform/advert/{id}

defaults: { \_controller: OCPlatformBundle:Advert:view }

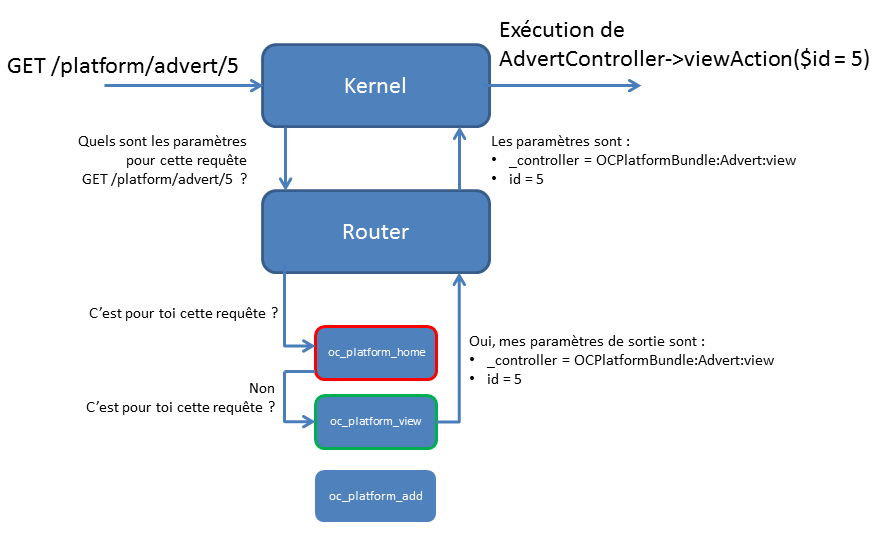
oc\_platform\_add:

path: /platform/add

defaults: { \_controller: OCPlatformBundle:Advert:add }

### Fonctionnement du routeur :

Pour chaque route on a donc une entrée : ‘path’ et une sortie ‘defaults’.



Pour chaque page, il est possible de voir toutes les routes que le routeur essaye une à une et celle qu’il utilise au final. C’est le **Profiler** qui s’occupe de gérer tout ça. On clique sur le nom de la route dans la barre d’outils ‘’oc\_platform\_home’’ si on est dans /platform.

Rappel sur les contrôleurs : OCPlatformBundle:Advert:view

Ici on indique que l’on utilise le Bundle OC/PlatformBundle, que l’on va chercher le contrôleur Advert et la méthode view.

## Les routes de base :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_home:

path: /platform

defaults:

\_controller: OCPlatformBundle:Advert:index

* Oc\_platform\_home n’a pas d’importance dans le fonctionnement, mais il doit être unique et pour être logique, respecter une certaine convention de nommage : oc\_platform en référence à OC/Platform.
* Path : /platform est l’URL sur laquelle la route s’applique.
* Defaults: \_controller : OCPlatformBundle :Advert :index est le paramètre de sortie de la route. Ici le seul contrôleur à appeler est mentionné, mais ce tableau defaults peut contenir d’autres paramètres.

### Créer une route avec des paramètres :

C’est la route 2 :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /platform/advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

Ici {id} prend n’importe quoi, ainsi la route sera sélectionnée pour : /platform/advert/5 par exemple. En revanche il doit être **OBLIGATOIREMENT PRESENT**.

Mais l’intérêt est d’avoir cet id en argument, on souhaite donc le récupérer.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// La route fait appel à OCPlatformBundle:Advert:view,

// on doit donc définir la méthode viewAction.

// On donne à cette méthode l'argument $id, pour

// correspondre au paramètre {id} de la route

public function viewAction($id)

{

// $id vaut 5 si l'on a appelé l'URL /platform/advert/5

// Ici, on récupèrera depuis la base de données

// l'annonce correspondant à l'id $id.

// Puis on passera l'annonce à la vue pour

// qu'elle puisse l'afficher

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

}

// ... et la méthode indexAction que nous avons déjà créée

}

En multipliant les paramètres on a comme route :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view\_slug:

path: /platform/{year}/{slug}.{format}

defaults:

\_controller: OCPlatformBundle:Advert:viewSlug

Et aussi dans le contrôleur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// On récupère tous les paramètres en arguments de la méthode

public function viewSlugAction($slug, $year, $format)

{

return new *Response*(

"On pourrait afficher l'annonce correspondant au

slug '".$slug."', créée en ".$year." et au format ".$format."."

);

}

}

L’ordre n’a pas d’importance dans la déclaration de l’URL, ce qui compte c’est le nom des variables. On peut séparer les variables par un point ou un slash, les variables ne doivent donc pas contenir de point.

## Les routes avancées :

Pour éviter que l’utilisateur ne raconte trop de connerie, on va spécifier les types de variables que l’on attend :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view\_slug:

path: /platform/{year}/{slug}.{format}

defaults:

\_controller: OCPlatformBundle:Advert:viewSlug

requirements:

year: \d{4}

format: html|xml

Avec requirements on spécifie ce qu’on attend dans year et format avec des expressions régulières.

### Paramètres facultatifs :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view\_slug:

path: /platform/{year}/{slug}.{format}

defaults:

\_controller: OCPlatformBundle:Advert:viewSlug

format: html

requirements:

year: \d{4}

format: html|xml

avec l’ajout de format : on a un paramètre facultatif. On peut ne pas renseigner le format il sera par défaut HTML dans ce cas.

defaults: { \_controller: OCPlatformBundle:Advert:viewSlug, format: html }

(Si on souhaite écrire sur une seule ligne).

### Utiliser des paramètres systèmes :

Si on a XML comme format, on va donc afficher du XML et on veut donc envoyer le Header avec le bon Content-type. On va donc utiliser des paramètres systèmes et le Kernel de Symfony va effectuer des actions supplémentaires :

**\_format :**

Quand il est utilisé, un header avec le Content-type est ajouté à la réponse.

Il faut aussi penser à mettre $\_format dans viewSlugAction(). Si on fait l’exemple, le navigateur gueule parce que ce n’est pas un format XML valide et bien formé.

**\_locale :**

Définit la langue avec laquelle affiché la page si on dispose de la page en plusieurs langues.

Penser à mettre un requirements pour éviter que les utilisateurs ne demandent une langue que l’on ne maitrise pas.

**\_controller :**

On ne le mettra jamais dans le path, mais c’est bien un paramètre système.

### Ajouter un préfixe lors de l’import de nos routes :

Au lieu de mettre /platfom au début de chaque path, on peut rajouter un prefix :

Ce coup-ci on modifie le fichier app/config/routing.yml

# app/config/routing.yml

oc\_platform:

resource: "@OCPlatformBundle/Resources/config/routing.yml"

prefix: /platform

## Générer des URL :

### Pourquoi générer des URL ?

Le routeur peut associer une route à une certaine URL mais aussi reconstruire l’URL à une certaine route.

Exemple :

On a une route nommée ‘’oc\_platform\_view’’ qui écoute l’URL /platform/advert/{id}, on peut décider de racourcir l’URL pour n’avoir que /platform/a/{id}.

### Comment générer des URL :

**Depuis le contrôleur :**

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function indexAction()

{

// On veut avoir l'URL de l'annonce d'id 5.

$url = $this->get('router')->generate(

'oc\_platform\_view', // 1er argument : le nom de la route

array('id' => 5) // 2e argument : les valeurs des paramètres

);

// $url vaut « /platform/advert/5 »

return new *Response*("L'URL de l'annonce d'id 5 est : ".$url);

}

}

Le nom de la route ici est référencé dans le Routeur :

oc\_platform\_view:

path: /advert/{id}

defaults: { \_controller: OCPlatformBundle:Advert:view }

Le deuxième argument est facultatif si on n’utilise pas d’arguments.

La méthode nous renverra donc : l'URL de l'annonce d'id est : /Symfony/web/app\_dev.php/platform/advert/5.

On récupère l’URL pour l’afficher.

**Si on veut une URL absolue :**

On doit définir le troisième argument de la méthode generate :

<?php

use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

$url = $this->get('router')->generate('oc\_platform\_home', array(), *UrlGeneratorInterface*::ABSOLUTE\_URL);

Ainsi on aura l’URL entière et pas uniquement la fin.

Comme notre contrôleur hérite de du contrôleur de Symfony on a accès à eux autres méthodes :

<?php

// Depuis un contrôleur

// Méthode longue

$url = $this->get('router')->generate('oc\_platform\_home');

// Méthode courte

$url = $this->generateUrl('oc\_platform\_home');

**Depuis une vue Twig :**

 On utilise path.

{# Dans une vue Twig, en considérant bien sûr

que la variable advert\_id est disponible #}

<a href="{{ path('oc\_platform\_view', { 'id': advert\_id }) }}">

Lien vers l'annonce d'id {{ advert\_id }}

</a>

Pour générer l’URL complète, on utilise url() à la place de path(). Pas besoin de troisème argument.

## Application : les routes de notre plateforme :

**Page d’accueil** : on veut pouvoir regarder les dernières annonces mais aussi les plus anciennes. On va donc définir {page}comme étant facultatif.

|  |  |
| --- | --- |
| /platform | page = 1 |
| /platform/1 | page = 1 |
| /platform/2 | page = 2 |

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_home:

path: /{page}

defaults:

\_controller: OCPlatformBundle:Advert:index

page: 1

requirements:

page: \d\*

**Page de visualisation d’une annonce :**

On rajoute juste un {id} pour choisir la bonne annonce.

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

requirements:

id: \d+

**Ajout, modification et suppression :**

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_add:

path: /add

defaults:

\_controller: OCPlatformBundle:Advert:add

oc\_platform\_edit:

path: /edit/{id}

defaults:

\_controller: OCPlatformBundle:Advert:edit

requirements:

id: \d+

oc\_platform\_delete:

path: /delete/{id}

defaults:

\_controller: OCPlatformBundle:Advert:delete

requirements:

id: \d+

# Les contrôleurs avec Symfony

Le chef d’orchestre, doit renvoyer une réponse !

## Le rôle du contrôleur :

**Retourner une réponse :**

Symfony s’inspire des concepts http. Il existe donc dans Symfony les classes *Request* et *Response*. Renvoyer une réponse = instancier une de ces classes.

Comme $response et faire un return $response.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function indexAction()

{

return new *Response*("Hello World !");

}

}

## L’objet Request :

### Les paramètres contenus dans les routes :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

requirements:

id: \d+

Le paramètre {id} est récupéré par la route et transformé en argument $id pour le contrôleur. Voici le code du contrôleur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// …

public function viewAction($id)

{

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

}

}

### Les paramètres hors routes :

C’est la méthode « à l’ancienne », imaginons qu’on ait : /platform/advert/5?tag=developer et qu’on aimerait récupérer la variable tag. On utilise l’objet *Request*.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request; // N'oubliez pas ce use !

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function viewAction($id, Request $request)

{

// Vous avez accès à la requête HTTP via $request

}

}

On va d’abords récupérer la requête. L’action est réalisée par le Kernel, il regarde si l’un des arguments de la méthode est typé avec Request, si c’est le cas il ajoute la requête aux arguments avant d’exécuter le contrôleur.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// …

// On injecte la requête dans les arguments de la méthode

public function viewAction($id, Request $request)

{

// On récupère notre paramètre tag

$tag = $request->query->get('tag');

return new *Response*(

"Affichage de l'annonce d'id : ".$id.", avec le tag : ".$tag

);

}

}

On a utilisé $request->query pour récupérer les paramètres de l’URL passés en GET, mais il y en a d’autre :

| **Type de paramètres** | **Méthode Symfony** | **Méthode traditionnelle** | **Exemple** |
| --- | --- | --- | --- |
| Variables d'URL | $request->query | $\_GET | $request->query->get('tag') |
| Variables de formulaire | $request->request | $\_POST | $request->request->get('tag') |
| Variables de cookie | $request->cookies | $\_COOKIE | $request->cookies->get('tag') |
| Variables de serveur | $request->server | $\_SERVER | $request->server->get('REQUEST\_URI') |
| Variables d'entête | $request->headers | $\_SERVER['HTTP\_\*'] | $request->headers->get('USER\_AGENT') |
| Paramètres de route | $request->attributes | n/a | On utilise $id dans les arguments de la méthode, mais vous pourriez également écrire  $request->attributes->get('id') |

Si on fait $request->query->get(‘sdf’) et que le paramètre sdf n’est pas dans l’URL, on aura une chaîne vide au lieu d’une erreur.

### Les autres méthodes de l’objet Request :

**Récupérer la méthode de la requête http :**

<?php

if ($request->isMethod('POST'))

{

// Un formulaire a été envoyé, on peut le traiter ici

}

**Savoir si la requête est une requête AJAX :**

<?php

if ($request->isXmlHttpRequest())

{

// C'est une requête AJAX, retournons du JSON, par exemple

}

**Les autres :**

[Sur le site de Symfony](http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html) :

**Décomposition de la construction d’un objet Response :**

Cas d’une erreur 404 :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// On modifie viewAction, car elle existe déjà

public function viewAction($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 à « Not Found » (erreur 404)

$response->setStatusCode(*Response*::HTTP\_NOT\_FOUND);

// On retourne la réponse

return $response;

}

}

### Réponses et vues :

Templating dispose d’un raccourci : renderResponse(). Elle prend en paramètre le nom du template et ses variables.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function viewAction($id, Request $request)

{

// On récupère notre paramètre tag

$tag = $request->query->get('tag');

// On utilise le raccourci : il crée un objet Response

// Et lui donne comme contenu le contenu du template

return $this->get('templating')->renderResponse(

'OCPlatformBundle:Advert:view.html.twig',

array('id' => $id, 'tag' => $tag)

);

}

}

Ou alors avec uniquement render() dont dispose le contrôleur comme raccourci :

<?php

public function viewAction($id, Request $request)

{

// On récupère notre paramètre tag

$tag = $request->query->get('tag');

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'id' => $id,

'tag' => $tag,

));

}

C’est ainsi qu’on gère la plupart des réponses, on ne manipule jamais directement les views.

Il faut bien sûr créer view.html.twig

{# src/OC/PlatformBundle/Resources/view/Advert/view.html.twig #}

<!DOCTYPE html>

<html>

<head>

<title>Affichage de l'annonce {{ id }}</title>

</head>

<body>

<h1>Hello Annonce n°{{ id }} !</h1>

<p>Tag éventuel : {{ tag }}</p>

</body>

</html>

### Réponse et redirection :

**Une redirection est une réponse http**. Il existe l’objet **RedirectResponse** qui étend l’objet Response en ajoutant l’entête **Location**.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\RedirectResponse; // N'oubliez pas ce use

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function viewAction($id)

{

$url = $this->get('router')->generate('oc\_platform\_home');

return new *RedirectResponse*($url);

}

}

Pour faire une redirection depuis un contrôleur : **redirect** : inutile d’intégrer le use RedirectResponse.

<?php

public function viewAction($id)

{

$url = $this->get('router')->generate('oc\_platform\_home');

return $this->redirect($url);

}

Ou de manière raccourcie :

<?php

public function viewAction($id)

{

return $this->redirectToRoute('oc\_platform\_home');

}

Ici on prend la route et non l’URL en argument. Pratique.

En passant intercept\_redirect à true dans app/config/config\_dev.yml. On peut intercepter les redirections.

### Changer le Content-type de la réponse :

Exemple : une requête AJAX qui renvoie un tableau au format JSON :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function viewAction($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 au navigateur

// que l'on renvoie du JSON et non du HTML

$response->headers->set('Content-Type', 'application/json');

return $response;

}

De manière raccourci :

<?php

use Symfony\Component\HttpFoundation\JsonResponse;

// ...

public function viewAction($id)

{

return new *JsonResponse*(array('id' => $id));

}

## Manipuler la session :

On les défini et récupère avec GET et SET :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function viewAction($id, Request $request)

{

// Récupération de la session

$session = $request->getSession();

// On récupère le contenu de la variable user\_id

$userId = $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*("<body>Je suis une page de test, je n'ai rien à dire</body>");

}

}

Dans le Profiler on peut voir la session et ses paramètres.

Il y a aussi une mémoire flash qui ne dure que le temps d’une page.

Exemple :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function viewAction($id)

{

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'id' => $id

));

}

// Ajoutez cette méthode :

public function addAction(Request $request)

{

$session = $request->getSession();

// Bien sûr, cette méthode devra réellement ajouter l'annonce

// Mais faisons comme si c'était le cas

$session->getFlashBag()->add('info', 'Annonce bien enregistrée');

// Le « flashBag » est ce qui contient les messages flash dans la session

// Il peut bien sûr contenir plusieurs messages :

$session->getFlashBag()->add('info', 'Oui oui, elle est bien enregistrée !');

// Puis on redirige vers la page de visualisation de cette annonce

return $this->redirectToRoute('oc\_platform\_view', array('id' => 5));

}

}

Le message flash s’affiche puis est détruit par la session.

La lecture des messages se fait dans la vue :

{# src/OC/PlatformBundle/Resources/view/Advert/view.html.twig #}

<!DOCTYPE html>

<html>

<head>

<title>Affichage de l'annonce {{ id }}</title>

</head>

<body>

<h1>Affichage de l'annonce n°{{ id }} !</h1>

<div>

{# On affiche tous les messages flash dont le nom est « info » #}

{% for message in app.session.flashbag.get('info') %}

<p>Message flash : {{ message }}</p>

{% endfor %}

</div>

<p>

Ici nous pourrons lire l'annonce ayant comme id : {{ id }}<br />

Mais pour l'instant, nous ne savons pas encore le faire, cela viendra !

</p>

</body>

</html>

En allant à [platform/add](http://localhost/Symfony/web/app_dev.php/platform/add) on est redirigé sur la page Avert/5 et on a bien les messages flashs.

## Le contrôleur de note plateforme :

Pour le mêment :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_home:

path: /{page}

defaults:

\_controller: OCPlatformBundle:Advert:index

page: 1

requirements:

page: \d\*

oc\_platform\_view:

path: /advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

requirements:

id: \d+

oc\_platform\_add:

path: /add

defaults:

\_controller: OCPlatformBundle:Advert:add

oc\_platform\_edit:

path: /edit/{id}

defaults:

\_controller: OCPlatformBundle:Advert:edit

requirements:

id: \d+

oc\_platform\_delete:

path: /delete/{id}

defaults:

\_controller: OCPlatformBundle:Advert:delete

requirements:

id: \d+

Et le contrôleur Advert :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController 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

// une page d'erreur 404 (qu'on pourra personnaliser plus tard d'ailleurs)

throw new *NotFoundHttpException*('Page "'.$page.'" inexistante.');

}

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

// Mais pour l'instant, on ne fait qu'appeler le template

return $this->render('OCPlatformBundle:Advert:index.html.twig');

}

public function viewAction($id)

{

// Ici, on récupérera l'annonce correspondante à l'id $id

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'id' => $id

));

}

public function addAction(Request $request)

{

// La gestion d'un formulaire est particulière, mais l'idée est la suivante :

// Si la requête est en POST, c'est que le visiteur a soumis le formulaire

if ($request->isMethod('POST')) {

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

$request->getSession()->getFlashBag()->add('notice', 'Annonce bien enregistrée.');

// Puis on redirige vers la page de visualisation de cettte annonce

return $this->redirectToRoute('oc\_platform\_view', array('id' => 5));

}

// Si on n'est pas en POST, alors on affiche le formulaire

return $this->render('OCPlatformBundle:Advert:add.html.twig');

}

public function editAction($id, Request $request)

{

// Ici, on récupérera l'annonce correspondante à $id

// Même mécanisme que pour l'ajout

if ($request->isMethod('POST')) {

$request->getSession()->getFlashBag()->add('notice', 'Annonce bien modifiée.');

return $this->redirectToRoute('oc\_platform\_view', array('id' => 5));

}

return $this->render('OCPlatformBundle:Advert:edit.html.twig');

}

public function deleteAction($id)

{

// Ici, on récupérera l'annonce correspondant à $id

// Ici, on gérera la suppression de l'annonce en question

return $this->render('OCPlatformBundle:Advert:delete.html.twig');

}

}

# Le moteur de template Twig

## Les templates Twig :

### Intérêt :

Permettent de séparer le code HTML du code XML, texte … Il sécurise les variables, il y a des fonctionnalités en plus.

### Page web, email …

Les templates doivent être utilisés partout, email, flux RSS …

### En pratique :

<?php

// Depuis un contrôleur

return $this->render('OCPlatformBundle:Advert:index.html.twig', array(

'var1' => $var1,

'var2' => $var2

));

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

<?php

// Depuis un contrôleur

$contenu = $this->renderView('OCPlatformBundle:Advert:email.txt.twig', array(

'var1' => $var1,

'var2' => $var2

));

// Puis on envoie l'e-mail, par exemple :

mail('moi@openclassrooms.com', 'Inscription OK', $contenu);

Et le template OCPlatformBundle:Advert:email.txt.twig contiendrait par exemple :

{# src/OC/PlatformBundle/Resources/views/Advert/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 !

**A savoir :**

* {{ … }}*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.

## Afficher des variables :

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

**Quelques filtres :**

| **Filtre** | **Description** | **Exemple Twig** |
| --- | --- | --- |
| [upper](http://twig.sensiolabs.org/doc/filters/upper.html) | Met toutes les lettres en majuscules. | {{ var|upper }} |
| [striptags](http://twig.sensiolabs.org/doc/filters/striptags.html) | Supprime toutes les balises XML. | {{ var|striptags }} |
| [date](http://twig.sensiolabs.org/doc/filters/date.html) | 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](http://twig.sensiolabs.org/doc/filters/format.html) | Insère des variables dans un texte, équivalent à [printf](https://php.net/printf). | {{ "Il y a %s pommes et %s poires"|format(153, nb\_poires) }} |
| [length](http://twig.sensiolabs.org/doc/filters/length.html) | 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 }} |

### Twig et la sécurité :

Twig protège automatiquement les variables. Pour ne plus échapper les balises <> de l’HTML, on utilise RAW : {{ ma\_variable\_html|raw }}

### Les variables globales :

Twig créer automatiquement {{ app }}, on a donc accès à :

| **Variable** | **Description** |
| --- | --- |
| {{ app.request }} | La requête « request » qu'on a vue 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.user }} | L'utilisateur courant, que nous verrons également plus loin dans ce cours. |

Pour éditer ses propres variables globales, on peut faire ainsi :

# app/config/config.yml

# …

twig:

# …

globals:

webmaster: moi-même

Ainsi {{ webmaster }} sera accessible dans toutes les vues.

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 :

# app/config/parameters.yml

parameters:

# …

app\_webmaster: moi-même

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

# app/config/config.yml

twig:

globals:

webmaster: %app\_webmaster%

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

## Structure et contrôle des expressions :

### Les structures de contrôle :

##### Condition : [{% if %}](http://twig.sensiolabs.org/doc/tags/if.html)

Exemple Twig :

{% if membre.age < 12 %}

Il faut avoir au moins 12 ans pour ce film.

{% elseif membre.age < 18 %}

OK bon film.

{% else %}

Un peu vieux pour voir ce film non ?

{% endif %}

Équivalent PHP :

<?php if($membre->getAge() < 12) { ?>

Il faut avoir au moins 12 ans pour ce film.

<?php } elseif($membre->getAge() < 18) { ?>

OK bon film.

<?php } else { ?>

Un peux vieux pour voir ce film non ?

<?php } ?>

##### Boucle : [{% for %}](http://twig.sensiolabs.org/doc/tags/for.html)

Exemple Twig :

<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 :

<select>

{% for valeur, option in liste\_options %}

<option value="{{ valeur }}">{{ option }}</option>

{% endfor %}

</select>

Équivalent 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 :

<?php

foreach($liste\_options as $valeur => $option) {

// …

}

##### Définition : [{% set %}](http://twig.sensiolabs.org/doc/tags/set.html)

Exemple Twig :

{% set foo = 'bar' %}

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

#### Les tests utiles

##### [Defined](http://twig.sensiolabs.org/doc/tests/defined.html)

Pour vérifier si une variable existe.

Exemple Twig :

{% if var is defined %} … {% endif %}

Équivalent PHP :

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

##### [Even](http://twig.sensiolabs.org/doc/tests/even.html) / [Odd](http://twig.sensiolabs.org/doc/tests/odd.html)

Pour tester si un nombre est pair / impair.

Exemple Twig :

{% for valeur in liste %}

<span class="{% if loop.index is even %}pair{% else %}

impair{% endif %}">

{{ valeur }}

</span>

{% endfor %}

Équivalent PHP :

<?php

$i = 0;

foreach($liste as $valeur) {

echo '<span class="';

echo $i % 2 ? 'impair' : 'pair';

echo '">'.$valeur.'</span>';

$i++;

}

## Hériter et inclure des templates :

Problème : un seul design mais on ne veut pas le répéter dans chaque template.

Pour résoudre le problème, on a un design père avec des blocs « trous », on définit ensuite dans des classes fils, des designs pour remplir les trous.

### La pratique :

Les templates pères sont appelés layout :

{# src/OC/PlatformBundle/Resources/views/layout.html.twig #}

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<title>{% block title %}OC Plateforme{% endblock %}</title>

</head>

<body>

{% block body %}

{% endblock %}

</body>

</html>

Template fils :

src/OC/PlatformBundle/Resources/views/Advert/index.html.twig :

{# src/OC/PlatformBundle/Resources/views/Advert/index.html.twig #}

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

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

{% block body %}

Notre plateforme est un peu vide pour le moment, mais cela viendra !

{% endblock %}

### Le nom du template père :

Pour la notation, on ne fait pas OCPlatformBundle :Controleur :layout.html.twig mais plutôt OCPlatformBundle ::layout.html.twig.

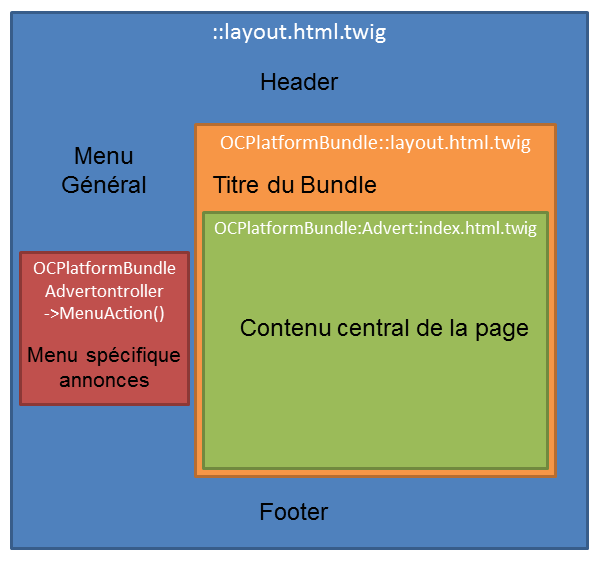
On peut mettre une valeur par défaut comme on a fait pour le titre.

* On peut récupérer cette valeur avec {{ parent() }}
* Si on ne redéfinit pas, on aura toujours une valeur.

### Organiser son code en triple héritage :

* Layout général : c'est le design de votre site, indépendamment de vos bundles. Il contient l'en-tête, le pied de page, 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 plateforme d'annonce, on pourrait afficher un menu particulier, rajouter « Annonces » dans le titre, etc.
* Template de page : il hérite du layout du bundle et contient le contenu central de votre page.

Le layout général se met dans App. app/Resources/views/layout.html.twig



### Héritage ou inclusion ?

Ajouter et modifier une annonce utilisent le même formulaire. On ne va pas faire de copier-coller du code, on utilise l’inclusion : On a nos deux templates OCPlatformBundle:Advert:add.html.twig et OCPlatformBundle:Advert:edit.html.twig qui héritent chacun de OCPlatformBundle::layout.html.twig.

L'affichage exact de ces deux templates diffère un peu, mais chacun d'eux inclut OCPlatformBundle:Advert:form.html.twig à l'endroit exact pour afficher le formulaire.

### La pratique :

L’include ( ):

{# src/OC/PlatformBundle/Resources/views/Advert/add.html.twig #}

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

{% block body %}

<h2>Ajouter une annonce</h2>

{{ include("OCPlatformBundle:Advert:form.html.twig") }}

<p>

Attention : cette annonce sera ajoutée directement

sur la page d'accueil après validation du formulaire.

</p>

{% endblock %}

Code du template inclus :

{# src/OC/PlatformBundle/Resources/views/Advert/form.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'annonce</h3>

{# 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>

### Inclusion de contrôleur :

Pour inclure un contrôleur on utilise {{ render() }}. Exemple avec le layout :

{# src/OC/PlatformBundle/Resources/views/layout.html.twig #}

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<title>{% block title %}OC Plateforme{% endblock %}</title>

</head>

<body>

<div id="menu">

{{ render(controller("OCPlatformBundle:Advert:menu")) }}

</div>

{% block body %}

{% endblock %}

</body>

</html>

On inclut un contrôleur quand on a besoin de quelque chose de précis qu’un template ne peut pas nous donner tout seul. On inclut le contrôleur qui génère les paramètres et appel ç son tour le template.

Il faut aussi ajouter la méthode menuAction dans le contrôleur.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

// ...

public function menuAction()

{

// On fixe en dur une liste ici, bien entendu par la suite

// on la récupérera depuis la BDD !

$listAdverts = array(

array('id' => 2, 'title' => 'Recherche développeur Symfony'),

array('id' => 5, 'title' => 'Mission de webmaster'),

array('id' => 9, 'title' => 'Offre de stage webdesigner')

);

return $this->render('OCPlatformBundle:Advert:menu.html.twig', array(

// Tout l'intérêt est ici : le contrôleur passe

// les variables nécessaires au template !

'listAdverts' => $listAdverts

));

}

}

Exemple de ce que pourrait être un template :

{# src/OC/PlatformBundle/Resources/views/Advert/menu.html.twig #}

{# Ce template n'hérite de personne,

tout comme le template inclus avec {{ include() }}. #}

<ul class="nav nav-pills nav-stacked">

{% for advert in listAdverts %}

<li>

<a href="{{ path('oc\_platform\_view', {'id': advert.id}) }}">

{{ advert.title }}

</a>

</li>

{% endfor %}

</ul>

## Application :

On travaille avec des variables vides au lieu de BDD pour le moment.

### Layout général :

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

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<title>{% block title %}OC Plateforme{% endblock %}</title>

{% block stylesheets %}

{# On charge le CSS de bootstrap depuis le site directement #}

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

{% endblock %}

</head>

<body>

<div class="container">

<div id="header" class="jumbotron">

<h1>Ma plateforme d'annonces</h1>

<p>

Ce projet est propulsé par Symfony,

et construit grâce au MOOC OpenClassrooms et SensioLabs.

</p>

<p>

<a class="btn btn-primary btn-lg" href="https://openclassrooms.com/courses/developpez-votre-site-web-avec-le-framework-symfony2">

Participer au MOOC »

</a>

</p>

</div>

<div class="row">

<div id="menu" class="col-md-3">

<h3>Les annonces</h3>

<ul class="nav nav-pills nav-stacked">

<li><a href="{{ path('oc\_platform\_home') }}">Accueil</a></li>

<li><a href="{{ path('oc\_platform\_add') }}">Ajouter une annonce</a></li>

</ul>

<h4>Dernières annonces</h4>

{{ render(controller("OCPlatformBundle:Advert:menu", {'limit': 3})) }}

</div>

<div id="content" class="col-md-9">

{% block body %}

{% endblock %}

</div>

</div>

<hr>

<footer>

<p>The sky's the limit © {{ 'now'|date('Y') }} 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/1.11.1/jquery.min.js"></script>

<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>

{% endblock %}

</body>

</html>

### Layout du Bundle :

{# src/OC/PlatformBundle/Resources/views/layout.html.twig #}

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

{% block title %}

Annonces - {{ parent() }}

{% endblock %}

{% block body %}

{# On définit un sous-titre commun à toutes les pages du bundle, par exemple #}

<h1>Annonces</h1>

<hr>

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

{% block ocplatform\_body %}

{% endblock %}

{% endblock %}

### Templates finaux :

{# src/OC/PlatformBundle/Resources/views/Advert/index.html.twig #}

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

{% block title %}

Accueil - {{ parent() }}

{% endblock %}

{% block ocplatform\_body %}

<h2>Liste des annonces</h2>

<ul>

{% for advert in listAdverts %}

<li>

<a href="{{ path('oc\_platform\_view', {'id': advert.id}) }}">

{{ advert.title }}

</a>

par {{ advert.author }},

le {{ advert.date|date('d/m/Y') }}

</li>

{% else %}

<li>Pas (encore !) d'annonces</li>

{% endfor %}

</ul>

{% endblock %}

La suite de notre appli dans le cours sur Twig.

# Installer un Bundle avec Composer

## C’est quoi ?

Il gère les dépendances. Exemple, si on utilise SwiftMailer pour envoyer nos mail, SwiftMailer est une dépendance de notre projet.

Les problèmes avec les bibliothèques externes et donc les dépendances :

* 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.

Pour que Composer sache où sont nos bibliothèques : si on regarde sur la bibliothèque de Symfony [ici](https://packagist.org/packages/symfony/symfony), on voit dans Require tout ce dont on a besoin pour le faire fonctionner.

Pour savoir comment ça fonctionne :

* 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.

## L’installation :

#### Installer Composer

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

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

Cette commande télécharge et exécute le fichier PHP suivant : [http://getcomposer.org/installer](https://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 :

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 :

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 un Bundle :

D’abord on crée un dossier test là où on a installé Composer. Dedans on crée un fichier json :

{

"require": {

"twig/extensions": "~1.0"

}

}

| **Valeur** | **Exemple** | **Description** |
| --- | --- | --- |
| Un numéro de version exact | "2.0.17" | Ainsi, Composer téléchargera cette version exacte. |
| Une plage de versions | ">=2.0,<2.6" | Ainsi, Composer téléchargera la version la plus à jour, à partir de la version 2.0 et en s'arrêtant avant la version 2.6. Par exemple, si les dernières versions sont 2.4, 2.5 et 2.6, Composer téléchargera la version 2.5. |
| Une plage de versions sémantique | "~2.1" | Ainsi, Composer téléchargera la version la plus à jour, à partir de la version 2.1 et en s'arrêtant avant la version 3.0. C'est une façon plus simple d'écrire ">=2.1,<3.0"  avec la syntaxe précédente. C'est la façon la plus utilisée pour définir la version des dépendances. |
| Un numéro de version avec joker « \* » | "2.0.\*" | Ainsi, Composer téléchargera 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. |
| Un nom de branche "dev-XXX" |  | C'est un cas un peu particulier, où Composer ira chercher la dernière modification d'une branche Git en particulier. N'utilisez cette syntaxe que pour les bibliothèques dont il n'existe pas de vraie version. Vous verrez assez souvent "dev-master", où "master" correspond à la branche principale d'un dépôt Git. |

Pour mettre à jour les dépendances on utilise update.

C:\wamp\www\test>php ../composer.phar update

Loading composer repositories with package information

Updating dependencies (including require-dev)

- Installing twig/twig (v1.23.1)

Downloading: 100%

- Installing twig/extensions (v1.3.0)

Downloading: 100%

Writing lock file

Generating autoload files

C:\wamp\www\test>

On peut vérifier dans le répertoire test/vendor :

* Composer a téléchargé la dépendance "twig/extensions" que l'on a définie, 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.

### Installer un Bundle :

* On se rend [ici](https://packagist.org), ici on recherche ‘’fixture’’, et clique sur le Bundle de Doctrine : doctrine/doctrine-fixtures-bundle. On regarde les prérequis, ici on prend la version 2.3 qui peut prendre Symfony 2 et 3.
* Une fois que l’on connait le Bundle ET sa version, on modifie composer.json pour y rajouter :

"incenteev/composer-parameter-handler": "~2.0",

"doctrine/doctrine-fixtures-bundle": "~2.3"

* On remet à jour avec : php composer.phar update.

Enfin, on modifie AppKernel pour déclarer le Bundle :

<?php

// app/AppKernel.php

// …

if (in\_array($this->getEnvironment(), array('dev', 'test'))) {

// ...

$bundles[] = new *Doctrine*\Bundle\FixturesBundle\DoctrineFixturesBundle();

}

// …

Ici on indique que l’on veut le Bundle seulement en phase de dev et de test, si on le veut aussi en prod’ il faut le mettre en dehors du if.

Composer s’occupe de tout, même de déclarer les namespaces pour l’autoload. On peut vérifier dans vendor/composer/autoload\_namespace.php.

On y voit la ligne dans app/autoload.php :

<?php

$loader = require \_\_DIR\_\_.'/../vendor/autoload.php';

// …

### Gérer l’autoload :

On modifie le composer.json et pas les autres.

// composer.json

{

// …

"autoload": {

"psr-4": {

"": "src/",

"VotreNamespace": "chemin/vers/la/bibliotheque"

},

"files": [ "app/AppKernel.php" ]

},

// …

}

Ensuite il faut exécuter la commande :

C:\wamp\www\Symfony>php ../composer.phar dump-autoload

Generating autoload files

# Services, théories et créations

## Pourquoi utiliser des services ?

En PHP, on utilise beaucoup d’objets : envoyer des e-mail, BDD …

En générale, chaque objet utilise d’autres objets, comment organiser l’instanciation et par lequel commencer ?

Chaque objet est défini en **tant** que service et le **conteneur** permet d’instancier.

### Qu’est-ce qu’un service ?

Un service est un objet PHP qui remplir une fonction associée à une configuration. Il est accessible n’importe où dans le code. Pour chaque fonctionnalité on crée un ou plusieurs services. Un service est une **CLASSE**.

La configuration du service est un moyen de l’enregistrer dans le conteneur de service.

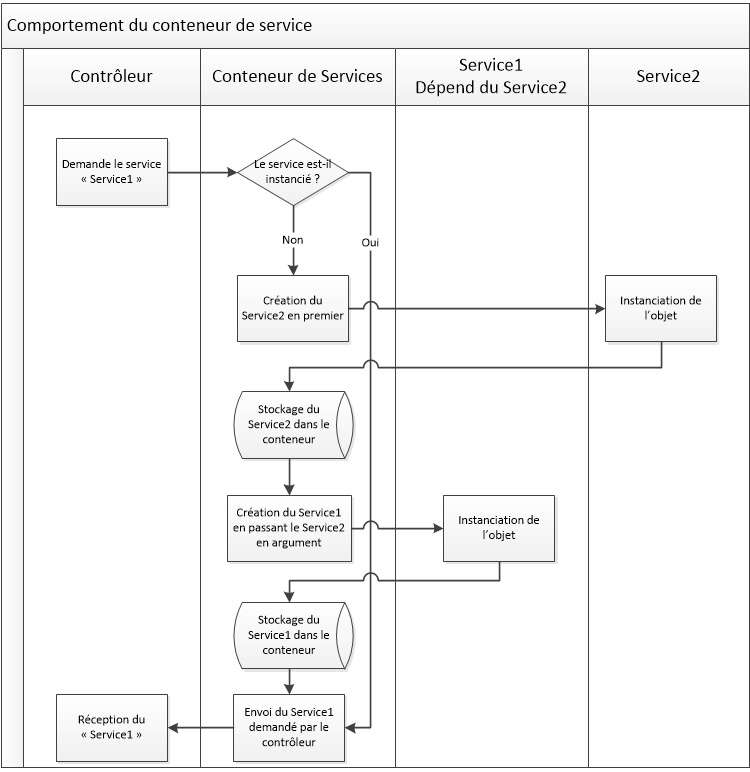
Prenons pour exemple l'envoi d'e-mails. Dans Symfony il existe le composant SwiftMailer qui permet de gérer les e-mails. Ce composant contient une classe nommée Swift\_Mailer qui envoie effectivement les e-mails. Symfony, qui intègre le composant SwiftMailer, définit déjà cette classe en tant que service mailer grâce à un peu de configuration. Le conteneur de service de Symfony peut donc accéder à la classe Swift\_Mailer  grâce au service mailer .

### Avantage de la POS (Programmation Orientée Service) :

Cela force à bien séparer chaque fonctionnalité. Comme chaque service ne remplit qu’une seule fonction, on peut facilement les réutiliser. On dit **Architecture Orienté Service**.

### Le conteneur des services :

C’est une sorte de super-objet qui gère tous les services. L’objectif est de simplifier au maximum la récupération des services. On l’appelle juste par son nom et le conteneur fait le reste.



Voilà ce qu’on aurait pu écrire dans ce code fictif si on avait dû écrire ce Container :

<?php

class Container

{

protected $service1 = null;

protected $service2 = null;

public function getService1()

{

if (null !== $this->service1) {

return $this->service1;

}

$service2 = $this->getService2();

$this->service1 = new *Service1*($service2);

return $this->service1;

}

public function getService2()

{

if (null !== $this->service2) {

return $this->service2;

}

$this->service2 = new *Service2*();

return $this->service2;

}

}

C’est comme ça que fait Symfony, preuve ici : var/cache/dev/appDevDebugProjectContainer.php

On a déjà utilisé un service de Templating, cette méthode a en fait :

<?php

protected function getTemplatingService()

{

$this->services['templating'] = $instance = new \Symfony\Bundle\TwigBundle\TwigEngine(

$this->get('twig'),

$this->get('templating.name\_parser'),

$this->get('templating.locator')

);

return $instance;

}

Ils se trouvent dans le cache, détruit à chaque fois donc. Comme ça si notre dépendance ajoute ou supprime une dépendance, le conteneur de service reflète le changement.

### Comment définir une dépendance entre service ?

L'idée est 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).

### Le partage des services :

Dans Symfony, chaque service est partagé : la classe du service est instanciée une seule fois par le conteneur.

## Utiliser un service en pratique :

### Récupérer un service :

Le composant Swiftmailer permet d’envoyer des emails. Présent par défaut sous le service **Mailer**. On utilise simplement un **GET**.

<?php

$container->get('mailer');

Pour avoir la liste des services disponibles, utilisez la commande php bin/console debug:container.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function indexAction()

{

// On a donc accès au conteneur :

$mailer = $this->container->get('mailer');

// On peut envoyer des e-mails, etc.

}

}

On a déjà utilisé $this->get() c’est parce que la classe Controller permet d’utiliser un raccourci, au lieu d’utiliser $this->controller->get( ).

## Créer un service simple :

La seule convention à respecter dans Symfony, c’est de **mettre notre classe dans un namespace correspondant au dossier où est le fichier**.

Par exemple, la classeOC\PlatformBundle\Antispam\OCAntispam doit se trouver dans le répertoiresrc/OC/PlatformBundle/Antispam/OCAntispam.php

On va créer un système anti-spam, il pourra être utilisable dans d’autres projets, on va le créer dans un Bundle à part donc. Pour que ce soit un simple exemple, on le laisse dans notre Bundle.

<?php

// src/OC/PlatformBundle/Antispam/OCAntispam.php

namespace OC\PlatformBundle\Antispam;

class OCAntispam

{

/\*\*

\* Vérifie si le texte est un spam ou non

\*

\* @param string $text

\* @return bool

\*/

public function isSpam($text)

{

return strlen($text) < 50;

}

}

Rien besoin de plus pour créer un service. On a respecté le namespace. Ici on considère que c’est un spam au-delà de 50 caractères.

### Création de la configuration du service :

Il faut maintenant la signaler au conteneur de service. (DependencyInjection/OCPlatformExtension.php) devrait être créé automatiquement, sinon se référer au cours.

Dans services.yml de Ressources/config, on ajoute ou créée :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.antispam:

class: OC\PlatformBundle\Antispam\OCAntispam

Dans cette configuration :

* oc\_platform.antispam est le nom de notre service fraîchement créé. De cette manière, le service sera accessible via $container->get('oc\_platform.antispam');. Essayez de respecter la convention en préfixant le nom de vos services par le nom du bundle, ici « oc\_platform ».
* class est un attribut obligatoire de notre configuration, il définit simplement le namespace complet de la classe du service. Cela indique au conteneur de services quelle classe instancier lorsqu'on lui demandera le service.

Pour définir un paramètre, la technique est la même que pour un service, dans le fichier services.yml:

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 $container->getParameter('nomParametre'); au lieu de get(). C'est d'ailleurs comme cela que vous pouvez récupérer les paramètres qui sont dans le fichier app/config/parameters.yml  comme les identifiants de votre base de données, etc.

### Utilisation des services :

Exemple :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// On récupère le service

$antispam = $this->container->get('oc\_platform.antispam');

// Je pars du principe que $text contient le texte d'un message quelconque

$text = '...';

if ($antispam->isSpam($text)) {

throw new \Exception('Votre message a été détecté comme spam !');

}

// Ici le message n'est pas un spam

}

}

## Créer un service avec des arguments :

Pour lui passer des arguments on utilise sa configuration.

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.antispam:

class: OC\PlatformBundle\Antispam\OCAntispam

arguments:

- "@mailer"

- %locale%

- 50

Les arguments peuvent être :

* Des valeurs normales en YAML (des booléens, des chaînes de caractères, des nombres, etc.) ;
* Des paramètres (définis dans leparameters.ymlpar exemple) : l'identifiant du paramètre est encadré de signes « **%** » :%nomDuParametre%;
* Des services : l'identifiant du service est précédé d'une arobase :@nomDuService.

Dans cet exemple, notre service utilise :

* @mailer: le service Mailer (pour envoyer des e-mails) ;
* %locale%: le paramètre locale (pour récupérer la langue, définit dans le fichierapp/config/parameters.yml) ;
* 50: et le nombre 50 (qu'importe son utilité !).

<?php

// src/OC/PlatformBundle/Antispam/OCAntispam.php

namespace OC\PlatformBundle\Antispam;

class OCAntispam

{

private $mailer;

private $locale;

private $minLength;

public function \_\_construct(\Swift\_Mailer $mailer, $locale, $minLength)

{

$this->mailer = $mailer;

$this->locale = $locale;

$this->minLength = (int) $minLength;

}

/\*\*

\* Vérifie si le texte est un spam ou non

\*

\* @param string $text

\* @return bool

\*/

public function isSpam($text)

{

return strlen($text) < $this->minLength;

}

}

isSpam est aussi modifié, maintenant on ne touche plus qu’à la config au lieu du code si on veut modifier des valeurs.

Comme on a fait avec Mailer, si on a besoin d’un service dans un service, on le passe en arguments dans les paramètres et Symfony se charge d’instancier la class.

# La couche métier : les entités :

On utilise un ORM : Object-Relation-manager qui se charge d’enregistrer les informations à notre place. Plus de phpmyadmin.

## Notion d’ORM :

Si on dispose d’une variable $variable contenant un objet User qui représente une personne qui s’inscrit.

On se contentera de faire :

$orm->save($utilisateur) ;

Il faut penser objet et non plus SQL.

Un objet à enregistrer s’appelle une **ENTITE** et on dit **PERSISTER** une entité et non enregistrer …

## Créer une première entité avec doctrine :

Ça ressemblerait à ça, avec son utilisation :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

class Advert

{

protected $id;

protected $content;

// Et bien sûr les getters/setters :

public function setId($id)

{

$this->id = $id;

}

public function getId()

{

return $this->id;

}

public function setContent($content)

{

$this->content = $content;

}

public function getContent()

{

return $this->content;

}

}

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function viewAction()

{

$advert = new *Advert*;

$advert->setContent("Recherche développeur Symfony3.");

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'advert' => $advert

));

}

}

### Les commentaires :

On utilise des commentaires pour ajouter des fonctionnalités, ils s’appellent **ANNOTATIONS**

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\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

\*/

class Advert

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="AUTO")

\*/

protected $id;

/\*\*

\* @ORM\Column(name="date", type="date")

\*/

protected $date;

/\*\*

\* @ORM\Column(name="title", type="string", length=255)

\*/

protected $title;

/\*\*

\* @ORM\Column(name="author", type="string", length=255)

\*/

protected $author;

/\*\*

\* @ORM\Column(name="content", type="text")

\*/

protected $content;

// Les getters

// Les setters

}

**Important à noter qu’on utilise /\*\* et non /\* !**

Ça s’appelle **MAPPER** l’objet. [Documentation DOctrine2](https://symfony.com/doc/master/doctrine.html)

### Générer une entité avec le générateur :

Pour générer on fait :

1. **Console : Php bin/console doctrine :generate :entity**
2. **On donne le sous la forme : OCPlatformBundle :Advert**
3. **Enter car on utilise les annotations (par défaut)**
4. **On entre le nom des champs**
5. **Leur type**
6. **Facultatif ?**
7. **Unique ?**
8. **On recommence pour les autres champs**
9. **Entrée quand on a fini**

Maintenant entity.php est créé avec les annotations déjà prêtes (le code est faisable à la main).

### Affiner l’entité avec de la logique métier :

Exemple :

Si on a une entité **COMMANDE** avec :

ants :

* ListeProduitsqui contient un tableau des produits de la commande ;
* AdresseLivraisonqui contient l'adresse où expédier la commande ;
* Datequi contient la date de la prise de la commande ;
* Etc.

Les attributs devront être **MAPPE** c’est-à-dire défini comme des colonnes pour l’ORM via des annotations.

Mais certains résultats demandent des calculs :

<?php

// Exemple :

class Commande

{

public function getPrixTotal()

{

$prix = 0;

foreach($this->getListeProduits() as $produit) {

$prix += $produit->getPrix();

}

return $prix;

}

}

Si on crée getListeProduits dans l’objet.

Nos entités étant des objets, on fait appel au constructeur et définir des variables :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* Advert

\*

\* @ORM\Table()

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

\*/

class Advert

{

// ...

public function \_\_construct()

{

// Par défaut, la date de l'annonce est la date d'aujourd'hui

$this->date = new \Datetime();

}

// ...

}

## Tout sur le mapping :

[Documentation sur le mapping](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html).

### Entity :

S’applique à une classe, elle définit un objet comme étant une entité, elle sera donc persistée par ORM.

@ORM\Entity

Il existe un paramètre facultatif qui précise le namespace.

@ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

### Class :

S’applique aussi à une classe. Facultatif il permet de préciser le nom :

@ORM\Table(name="oc\_advert")

### Column :

Définit les caractéristiques de la colonne concernée.

Plusieurs types :

| **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 booléennestrueetfalse. |
| 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 utilisantserialize/unserialize. |
| array | CLOB | array | Stocke un tableau PHP en utilisantserialize/unserialize. |
| float | FLOAT | double | Tous les nombres à virgule. Attention, fonctionne uniquement sur les serveurs dont la locale utilise un point comme séparateur. |

@ORM\Column(type="string")

Plusieurs paramètres facultatifs :

| **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 colonnestring. |
| 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 desNULL. |
| 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 colonnedecimal. |
| scale | 0 | Définit le *scale* d'un nombre à virgule, c'est-à-dire le nombre de chiffres après la virgule. Applicable uniquement sur un type de colonnedecimal. |

Pour définir plusieurs options en même temps, il faut simplement les séparer avec une virgule. Par exemple, pour une colonne « e-mail » enstring255 et unique, il faudra faire :

@ORM\Column(type="string", length=255, unique=true)

# Manipuler ses entités avec doctrine :

## Matérialiser les tables en BDD :

Pour créer la table :

**Php bin/console doctrine :database :create**

C:\wamp\www\Symfony>php bin/console doctrine:database:create

Created database `symfony` for connection named default

Ensuite on génère les tables à l’intérieur :

php bin/console doctrine:schema:update --dump-sql

On fait le point entre ce qu’on a dans nos entités et l’état de la BDD. Ici on fait le point sur ce qui va être fait **MAIS RIEN N’EST FAIT !**

Il faut forcer l’action :

C:\wamp\www\Symfony>php bin/console doctrine:schema:update --force

Updating database schema...

Database schema updated successfully! "1" query were executed

Maintenant la table est mise à jour.

### Modifier une entité :

Pour modifier une entité il suffit de créer un attribut et de lui rattacher l’annotation correspondante (ligne 18 à 21).

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* Advert

\*

\* @ORM\Table()

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

\*/

class Advert

{

// ... les autres attributs

/\*\*

\* @ORM\Column(name="published", type="boolean")

\*/

private $published = true;

// ...

}

Ensuite, pour les getters et les setters, on peut les faire à la main ou utiliser le générateur :

C:\wamp\www\Symfony>php bin/console doctrine:generate:entities OCPlatformBundle:Advert

Generating entity "OC\PlatformBundle\Entity\Advert"

  > backing up Advert.php to Advert.php~

  > generating OC\PlatformBundle\Entity\Advert

**Au lieu d’utiliser entity qui génère tout une entité, on utilise entities.**

Ensuite on regarde si on a bien le SQL correspondant et on l’applique (2 commandes différentes).

## Enregistrer ses entités avec l’Entity Manager :

### Les services de Doctrine2 :

**Rappel :** un service est accessible dans tout le code.

<?php

$doctrine = $this->get('doctrine');

L﻿﻿a classe Controller de Symfony intègre un raccourci. Il fait exactement la même chose, mais est plus joli et permet l'autocomplétion :

<?php

$doctrine = $this->getDoctrine();

Il est possible d’utiliser plusieurs connexions BDD avec $doctrine->getConnection($name) ;

Il dispose aussi de :

**$doctrine->getManager($name) ;**

Qui récupère un ORM à partir de son nom.

### EntityManager :

<?php

$em = $this->getDoctrine()->getManager();

Mais sachez que, comme tout service qui se respecte, vous pouvez y accéder directement via :

<?php

$em = $this->get('doctrine.orm.entity\_manager');

L’EntityManager permet de dire à Doctrine de dire « pesiste moi ça ».

### Les repositories :

<?php

$em = $this->getDoctrine()->getManager();

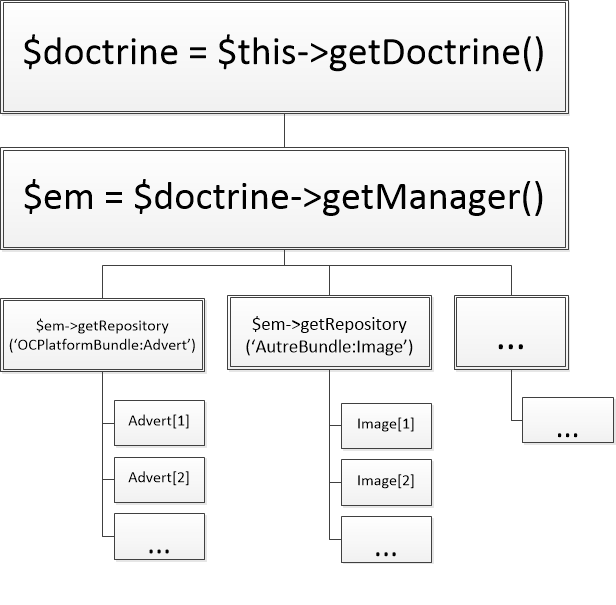
$advertRepository = $em->getRepository('OCPlatformBundle:Advert');

Il existe un repositorie par entité. Il faut toujours préciser son nom.

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 : 'OC\PlatformBundle\Entity\Advert'.
* Soit en utilisant le raccourci Nom\_du\_bundle:Nom\_de\_l'entité. Pour notre exemple, c'est donc 'OCPlatformBundle:Advert'. C'est un raccourci qui fonctionne partout dans Doctrine.

Attention, ce raccourci ne fonctionne que si vous avez mis vos entités dans le namespace Entitydans votre bundle.



### Enregistrer dans la base :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// Création de l'entité

$advert = new *Advert*();

$advert->setTitle('Recherche développeur Symfony.');

$advert->setAuthor('Alexandre');

$advert->setContent("Nous recherchons un développeur Symfony débutant sur Lyon. Blabla…");

// 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

$em->persist($advert);

// Étape 2 : On « flush » tout ce qui a été persisté avant

$em->flush();

// Reste de la méthode qu'on avait déjà écrit

if ($request->isMethod('POST')) {

$request->getSession()->getFlashBag()->add('notice', 'Annonce bien enregistrée.');

// Puis on redirige vers la page de visualisation de cettte annonce

return $this->redirectToRoute('oc\_platform\_view', array('id' => $advert->getId()));

}

// Si on n'est pas en POST, alors on affiche le formulaire

return $this->render('OCPlatformBundle:Advert:add.html.twig');

}

}

* La ligne 15 permet de créer l'entité, et les lignes 16 à 18 de renseigner ses attributs ;
* La ligne 23 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 précédemment (il fait donc des INSERT INTO  & Cie) ;
* Ligne 36, notre Advert étant maintenant enregistré en base de données grâce au flush(), Doctrine2 lui a attribué un id ! On peut donc utiliser $advert->getId()  dans la génération de la route, et non un nombre fixe comme précédemment.

### Doctrine utilise les transactions :

On a fait $em->persist() et $em->flush() à cause des transactions. Si on fait un forum et qu’on enregistre un un sujet mais pas le message à cause d’une erreur … la BDD est cassée à cause des relations.

**Avec une transaction, on enregistre en même temps**. SI une échoue, les autres sont annulés.

Du coup : persist dit : garde en mémoire jusqu’au Flush, au Flush on enregistre.

### Doctrin simplifie la vie :

Persist() traite indifférament chaque requette : celle que l’on vient de créer comme celle que l’on a récupéré avec le repositorie. On ne se soucie pas des update et insert into.

<?php

// Depuis un contrôleur

$em = $this->getDoctrine()->getManager();

// On crée une nouvelle annonce

$advert1 = new *Advert*;

$advert1->setTitle('Recherche développeur.');

$advert1->setContent("Pour mission courte");

// Et on le persiste

$em->persist($advert1);

// On récupère l'annonce 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

$advert2 = $em->getRepository('OCPlatformBundle:Advert')->find(5);

// On modifie cette annonce, en changeant la date à la date d'aujourd'hui

$advert2->setDate(new \Datetime());

// Ici, pas besoin de faire un persist() sur $advert2. En effet, comme on a

// récupéré cette annonce 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 deux changements à la base de données :

// Un INSERT INTO pour ajouter $advert1

// Et un UPDATE pour mettre à jour la date de $advert2

$em->flush();

**Inutile de faire un persist(entity) quand on récupère une entité avec Doctrine. Quand on crée un objet, Doctrine n’est pas sensé savoir que c’est une entité que l’on va ajouter, alors que quand on récupère une entité, on sait déjà que c’est une entité.**

Il ne fera pas d’UPDATE non plus si on ne modifie rien.

### Les autres méthodes de l’EntityManager :

[Documentation officiel](http://www.doctrine-project.org/api/orm/2.5/class-Doctrine.ORM.EntityManager.html) pour toutes les voir.

**detach($entite)**  annule le persist()  effectué sur l'entité en argument. Au prochain flush() , aucun changement ne sera donc appliqué à l'entité. Voici un exemple :

<?php

$em->persist($advert);

$em->persist($comment);

$em->detach($advert);

$em->flush(); // Enregistre $comment mais pas $advert

**clear($nomEntite)**  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. Si clear()  est appelé sans argument, cela revient à faire un detach()  sur toutes les entités d'un coup. Voici un exemple :

<?php

$em->persist($advert);

$em->persist($comment);

$em->clear();

$em->flush(); // N'exécutera rien, car les deux persists sont annulés par le clear

**contains($entite)** 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 :

<?php

$em->persist($advert);

var\_dump($em->contains($advert)); // Affiche true

var\_dump($em->contains($comment)); // Affiche false

**refresh($entite)** 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 :

<?php

$advert->setTitle('Un nouveau titre');

$em->refresh($advert);

var\_dump($advert->getTitle()); // Affiche « Un ancien titre »

**remove($entite)** supprime l'entité donnée en argument de la base de données. Effectif au prochain flush(). Voici un exemple :

<?php

$em->remove($advert);

$em->flush(); // Exécute un DELETE sur $advert

## Récupérer ses entités avec un repositories :

D’abords on récupère le repositorie :

<?php

// Depuis un contrôleur

$repository = $this->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

;

Puis,﻿﻿ depuis ce repository, il faut utiliser la méthode find($id) qui retourne l'entité correspondant à l'id $id. Exemple :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

public function viewAction($id)

{

// On récupère le repository

$repository = $this->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

;

// On récupère l'entité correspondante à l'id $id

$advert = $repository->find($id);

// $advert est donc une instance de OC\PlatformBundle\Entity\Advert

// ou null si l'id $id n'existe pas, d'où ce if :

if (null === $advert) {

throw new *NotFoundHttpException*("L'annonce d'id ".$id." n'existe pas.");

}

// Le render ne change pas, on passait avant un tableau, maintenant un objet

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'advert' => $advert

));

}

}

Il existe une autre syntaxe pour faire la même chose directement depuis l'EntityManager. Il s'agit de la méthode find de l'EntityManager, et non du Repository :

<?php

// Depuis un contrôleur

$advert = $this->getDoctrine()

->getManager()

->find('OCPlatformBundle:Advert', $id)

;

**RAPPEL :**

**Pour créer une entité : php bin/console doctrine :generate :entity**

**Pour créer une BDD : hp bin/console doctrine :database :create**

**Voir requête : php bin/console doctrine:schema:update --dump-sql**

**Forcer requete : php bin/console doctrine:schema:update --force**

**$em = $this->getDoctrine()->getManager**

**$advert1 = new Advert() ;**

**$advert1->setTitle(…)**

**$advert1->setContent(…)**

**$em->persist($advert1) ;**

**$advert2 = $em->getRepository(‘OCPlatformBundle :Advert’)->find(5) ;**

**$advert2->setDate(new \Datetime()) ;**

**$em->flush**

# Relation entre entité et Doctrine

## Présentation :

Il y a plusieurs façons de lier les entités entre elles. **OneToOne**, **OneToMany**, **ManyToMany**.

### Notion de propriétaire et d’inverse :

Il y a toujours une relation propriétaire et un inverse entre deux entités. **L’entité propriétaire est celle qui contient la référence à l’autre entité**.

Exemple :

Si on a une table *comment* et une table *Advert* dans *comment* on aura une colonne Advert\_id. La table *comment* est donc **propriétaire** de la table *Advert* car c’est elle qui fait la liaison.

C’est un exemple, on doit penser objet et non plus BDD. On ne va pas créer cette colonne ni toucher au phpmyadmin.

### Notion d’unidirectionnalité et bidirectionnalité :

On va traiter le cas de l’unidirectionnalité, c’est-à-dire qu’on va pouvoir écrire : $entitéPropriétaire->getEntiteInverse() ; Ici $comment->getAdvert() ;

Mais on ne peut pas faire $EntiteInverse->getEntitePropriétaire() ;

Doctrine utilise le **Lazy Loading,** il ne charge les relations que si on souhaite réélement y accéder. Au moment où l’on fait $Advert->getComments() ;

## Relation OneToOne :

Si on a une entité IMAGE, on dit que chaque annonce est liée à une image, et une image est liée à chaque annonce : 1To1.

Création de l’entité Image avec deux attributs : $url et $alt.

<?php

// src/OC/PlatformBundle/Entity/Image

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Table(name="oc\_image")

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\ImageRepository")

\*/

class Image

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @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;

}

D’abords on fait php bin/console doctrine :generate :entity.

Puis php bin/console doctrine :generate :entities OCPlatformBundle :Image pour générer les setters.

### Définition de la relation :

**Entité propriétaire,Advert :**

<?php

// src/OC/PlatformBundle/Entity/Advert.php

/\*\*

\* @ORM\Entity

\*/

class Advert

{

/\*\*

\* @ORM\OneToOne(targetEntity="OC\PlatformBundle\Entity\Image", cascade={"persist"})

\*/

private $image;

// …

}

**Entité inverse,Image:**

<?php

// src/OC/PlatformBundle/Entity/Image

/\*\*

\* @ORM\Entity

\*/

class Image

{

// Nul besoin d'ajouter une propriété ici

// …

}

On met Advert en propriétaire car en temps normal c’est un Advert « possède » une image, et non l’inverse.

On peut donc faire $Advert->getImage() mais pas l’inverse car c’est Advert qui est propriétaire !

@ORM\OneToOne(targetEntity="OC\PlatformBundle\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 parlera un peu plus loin.

### Rendre une relation non-facultative :

Par défaut la relation est facultative, on peut avoir un Advert sans image, pour changer ça, on doit faire un JoinColumn.

/\*\*

\* @ORM\OneToOne(targetEntity="OC\PlatformBundle\Entity\Image", cascade={"persist"})

\* @ORM\JoinColumn(nullable=false)

\*/

private $image;

### Les opérations de cascade :

**Cascade permet d’étendre l’opération sur les éléments liés. Par exemple, il on supprime un Advert et que l’image ne sert QUE à l’advert on le supprime aussi. Ici, on a fait une cascade sur PERSIST(), comme on fait un persist sur l’entité avant de FLUSH, on veut qu’il le fasse aussi sur l’image.**

Quand on fait un php bin/console doctrine :generate :update –force après un OneToOne, si on refait un php bin/console doctrine :generate :entities OCPlatformBundle :Advert, on a le getter et setter de Image :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

\*/

class Advert

{

/\*\*

\* @ORM\OneToOne(targetEntity="OC\PlatformBundle\Entity\Image", cascade={"persist"})

\*/

private $image;

// Vos autres attributs…

public function setImage(Image $image = null)

{

$this->image = $image;

}

public function getImage()

{

return $this->image;

}

// Vos autres getters/setters…

}

On a =null car il peut être facultatif.

Exemple :

Pour récupérer l’URL de l’image d’un Advert :

<?php

$image = $advert->getImage();

$url = $image->getUrl();

// Ou bien sûr en une seule ligne :

$url = $advert->getImage()->getUrl();

 une colonneimage\_ida bien été ajouté à la tableadvert. Cependant, ne confondez surtout par cette colonneimage\_idavec notre attributimage, et gardez bien ces deux points en tête :  
1/ L'objet Advert*ne* contient *pas* d'attributimage\_id.  
2/ L'attributimage de l'objetAdvert *ne* contient *pas* l'id de l'Imageliée, il contient une instance de la classeOC\PlatformBundle\Entity\Imagequi, *elle*, contient un attributid.

N'allez donc jamais m'écrire$advert->getImageId(), pour récupérer l'id de l'image liée, il faut d'abord récupérer l'Image elle-même puis son id, comme ceci :$advert->getImage()->getId(). Et ne faites surtout pas la confusion : une entité n'est pas une table.

Exemple d’utilisation :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

// N'oubliez pas ces use

use OC\PlatformBundle\Entity\Advert;

use OC\PlatformBundle\Entity\Image;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// Création de l'entité Advert

$advert = new *Advert*();

$advert->setTitle('Recherche développeur Symfony.');

$advert->setAuthor('Alexandre');

$advert->setContent("Nous recherchons un développeur Symfony débutant sur Lyon. Blabla…");

// Création de l'entité Image

$image = new *Image*();

$image->setUrl('http://sdz-upload.s3.amazonaws.com/prod/upload/job-de-reve.jpg');

$image->setAlt('Job de rêve');

// On lie l'image à l'annonce

$advert->setImage($image);

// On récupère l'EntityManager

$em = $this->getDoctrine()->getManager();

// Étape 1 : On « persiste » l'entité

$em->persist($advert);

// 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

}

Dans la vue on affiche enfin l’image :

{# src/OC/PlatformBundle/Resources/view/Advert/view.html.twig #}

{# On vérifie qu'une image soit bien associée à l'annonce #}

{% if advert.image is not null %}

<img src="{{ advert.image.url }}" alt="{{ advert.image.alt }}">

{% endif %}

Exemple qui modifirait l’image :

<?php

// Dans un contrôleur, celui que vous voulez

public function editImageAction($advertId)

{

$em = $this->getDoctrine()->getManager();

// On récupère l'annonce

$advert = $em->getRepository('OCPlatformBundle:Advert')->find($advertId);

// On modifie l'URL de l'image par exemple

$advert->getImage()->setUrl('test.png');

// On n'a pas besoin de persister l'annonce ni l'image.

// Rappelez-vous, ces entités sont automatiquement persistées car

// on les a récupérées depuis Doctrine lui-même

// On déclenche la modification

$em->flush();

return new *Response*('OK');

}

## Relation ManyToOne :

Une relation ManyToOne permet à une entité d’être lié à plusieurs entités. Comme plusieurs candidats pour une annonce.

On va créer une entité Application :

<?php

// src/OC/PlatformBundle/Entity/Application.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Table(name="oc\_application")

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\ApplicationRepository")

\*/

class Application

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="AUTO")

\*/

private $id;

/\*\*

\* @ORM\Column(name="author", type="string", length=255)

\*/

private $author;

/\*\*

\* @ORM\Column(name="content", type="text")

\*/

private $content;

/\*\*

\* @ORM\Column(name="date", type="datetime")

\*/

private $date;

public function \_\_construct()

{

$this->date = new \Datetime();

}

public function getId()

{

return $this->id;

}

public function setAuthor($author)

{

$this->author = $author;

return $this;

}

public function getAuthor()

{

return $this->author;

}

public function setContent($content)

{

$this->content = $content;

return $this;

}

public function getContent()

{

return $this->content;

}

public function setDate(\Datetime $date)

{

$this->date = $date;

return $this;

}

public function getDate()

{

return $this->date;

}

}

### Définir la relation :

Comme pour OneToOne on fait :

<?php

// src/OC/PlatformBundle/Entity/Application.php

/\*\*

\* @ORM\Entity

\*/

class Application

{

/\*\*

\* @ORM\ManyToOne(targetEntity="OC\PlatformBundle\Entity\Advert")

\* @ORM\JoinColumn(nullable=false)

\*/

private $advert;

// …

}

**Entité inverse,Advert :**

<?php

// src/OC/PlatformBundle/Entity/Advert.php

/\*\*

\* @ORM\Entity

\*/

class Advert

{

// Nul besoin de rajouter de propriété, ici

// …

}

Ici C’est many qui est propriétaire, on n’a pas le choix, du coup c’est application qui contient Avert\_id. On a laissé nullable = false pour éviter les candidatures spontannées, à adapter en fonction des besoins. On a donc plusieurs candidats pour une même application.

Après on génères les getters et setters puis on generate :schema :update –force.

Il n’y a pas le « null » dans setAdvert parce qu’il est obligatoire.

## Relation ManyToMany :

Une catégorie possède plusieurs annonces. Aussi, une annonce peut correspondre à plusieurs catégories.

Pour faire cela il faut une table intérmediaire, avec categorie\_id et advert\_id pour faire le lien entre les deux. Ici on ne la voit pas, c’est Doctrine qui fait ça tout seul.

Encore une fois, pour être sûrs que l'on parle bien de la même chose, créez cette entitéCategory avec au moins un attributname. Voici la mienne :

<?php

// src/OC/PlatformBundle/Entity/Category.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Entity

\*/

class Category

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="AUTO")

\*/

private $id;

/\*\*

\* @ORM\Column(name="name", type="string", length=255)

\*/

private $name;

public function getId()

{

return $this->id;

}

public function setName($name)

{

$this->name = $name;

}

public function getName()

{

return $this->name;

}

}

On choisis Advert comme propriétaire et non Category. On recherche plus souvent les catégories en fonction de l’advert et non l’inverse.

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

// N'oubliez pas ce use

use Doctrine\Common\Collections\ArrayCollection;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

\*/

class Advert

{

/\*\*

\* @ORM\ManyToMany(targetEntity="OC\PlatformBundle\Entity\Category", cascade={"persist"})

\*/

private $categories;

// … vos autres attributs

// Comme la propriété $categories doit être un ArrayCollection,

// On doit la définir dans un constructeur :

public function \_\_construct()

{

$this->date = new \Datetime();

$this->categories = new *ArrayCollection*();

}

// Notez le singulier, on ajoute une seule catégorie à la fois

public function addCategory(Category $category)

{

// Ici, on utilise l'ArrayCollection vraiment comme un tableau

$this->categories[] = $category;

}

public function removeCategory(Category $category)

{

// Ici on utilise une méthode de l'ArrayCollection, pour supprimer la catégorie en argument

$this->categories->removeElement($category);

}

// Notez le pluriel, on récupère une liste de catégories ici !

public function getCategories()

{

return $this->categories;

}

// … vos autres getters/setters

}

Si vous avez été vérifié dans PhpMyAdmin, vous aurez noté que la table créée s'appelleadvert\_category, or depuis le début nous préfixons nos noms de table avec "oc". Comment le faire également pour la table de jointure ? Il faut pour cela rajouter l'annotationJoinTable, qui permet de personnaliser un peu cette table, voici comment changer son nom par exemple :

<?php

/\*\*

\* @ORM\ManyToMany(targetEntity="OC\PlatformBundle\Entity\Category", cascade={"persist"})

\* @ORM\JoinTable(name="oc\_advert\_category")

\*/

private $categories;

### Remplir la BDD avec les fixtures :

On va faire une fixture Doctrine, qui va nous permettre d’utiliser le Bundle qu’on a installé avec Composer.

On va déjà créer le fichier de fixture pour l’entité Category. Ils se trouvent dans le répertoire DataFixtures/ORM (ou ODM pour des documents).

<?php

// src/OC/PlatformBundle/DataFixtures/ORM/LoadCategory.php

namespace OC\PlatformBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;

use Doctrine\Common\Persistence\ObjectManager;

use OC\PlatformBundle\Entity\Category;

class LoadCategory 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

$names = array(

'Développement web',

'Développement mobile',

'Graphisme',

'Intégration',

'Réseau'

);

foreach ($names as $name) {

// On crée la catégorie

$category = new *Category*();

$category->setName($name);

// On la persiste

$manager->persist($category);

}

// On déclenche l'enregistrement de toutes les catégories

$manager->flush();

}

}

Les fixtures permettent de remplir les bases de données avec un jeu de donnée que nous allons définir. Maintenant on peut ces données dans la BDD :

C:\wamp\www\Symfony>php bin/console doctrine:fixtures:load

Careful, database will be purged. Do you want to continue Y/N ?y

  > purging database

  > loading OC\PlatformBundle\DataFixtures\ORM\LoadCategory

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--appendà 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…

On va devoir recréer l’annocne avec add.

Exemple d’utilisation :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

// …

public function editAction($id, Request $request)

{

$em = $this->getDoctrine()->getManager();

// On récupère l'annonce $id

$advert = $em->getRepository('OCPlatformBundle:Advert')->find($id);

if (null === $advert) {

throw new *NotFoundHttpException*("L'annonce d'id ".$id." n'existe pas.");

}

// La méthode findAll retourne toutes les catégories de la base de données

$listCategories = $em->getRepository('OCPlatformBundle:Category')->findAll();

// On boucle sur les catégories pour les lier à l'annonce

foreach ($listCategories as $category) {

$advert->addCategory($category);

}

// Pour persister le changement dans la relation, il faut persister l'entité propriétaire

// Ici, Advert est le propriétaire, donc inutile de la persister car on l'a récupérée depuis Doctrine

// Étape 2 : On déclenche l'enregistrement

$em->flush();

// … reste de la méthode

}

}

Exemple pour retirer une annonce :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

// …

public function deleteAction($id)

{

$em = $this->getDoctrine()->getManager();

// On récupère l'annonce $id

$advert = $em->getRepository('OCPlatformBundle:Advert')->find($id);

if (null === $advert) {

throw new *NotFoundHttpException*("L'annonce d'id ".$id." n'existe pas.");

}

// On boucle sur les catégories de l'annonce pour les supprimer

foreach ($advert->getCategories() as $category) {

$advert->removeCategory($category);

}

// Pour persister le changement dans la relation, il faut persister l'entité propriétaire

// Ici, Advert est le propriétaire, donc inutile de la persister car on l'a récupérée depuis Doctrine

// On déclenche la modification

$em->flush();

// ...

}

}

Afficher les annonces dans la vue :

{# src/OC/PlatformBundle/Resources/view/Advert/view.html.twig #}

{% if not advert.categories.empty %}

<p>

Cette annonce est parue dans les catégories suivantes :

{% for category in advert.categories %}

{{ category.name }}{% if not loop.last %}, {% endif %}

{% endfor %}

</p>

{% endif %}

## Relation ManyToMany avec attribut :

Exemple, sur un site de commerce, on a **PRODUIT** et **COMMANDE** et un produit peut être dans plusieurs commande, et une commande peut contenir plusieurs produits. Il nous manque alors le nombre d’exemplaire ! C’est un attribut qui existe dans la relation entre ces deux entités, il n’appartient ni à l’une ni à l’autre.

Pour cela il faut créer une entité intermediaire, par exemple CommandeProduit. On fait ensuite **Commande** OneToMany **CommandeProduit** ManyToOne **Produit**.

Dans notre cas on a un niveau de compétence à mettre en fonction d’une compétence, soit :

* Advert <=>Commande
* AdvertSkill <=>CommandeProduit
* Skill <=>Produit

Et donc :Advert One-To-Many  AdvertSkill Many-To-One  Skill.

On va créer **SKILL** :

<?php

// src/OC/PlatformBundle/Entity/Skill.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Entity

\* @ORM\Table(name="oc\_skill")

\*/

class Skill

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="AUTO")

\*/

private $id;

/\*\*

\* @ORM\Column(name="name", type="string", length=255)

\*/

private $name;

public function getId()

{

return $this->id;

}

public function setName($name)

{

$this->name = $name;

}

public function getName()

{

return $this->name;

}

}

On créer aussi notre entité de relation entre les deux autres entités :

<?php

// src/OC/PlatformBundle/Entity/AdvertSkill.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Entity

\* @ORM\Table(name="oc\_advert\_skill")

\*/

class AdvertSkill

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="AUTO")

\*/

private $id;

/\*\*

\* @ORM\Column(name="level", type="string", length=255)

\*/

private $level;

/\*\*

\* @ORM\ManyToOne(targetEntity="OC\PlatformBundle\Entity\Advert")

\* @ORM\JoinColumn(nullable=false)

\*/

private $advert;

/\*\*

\* @ORM\ManyToOne(targetEntity="OC\PlatformBundle\Entity\Skill")

\* @ORM\JoinColumn(nullable=false)

\*/

private $skill;

// ... vous pouvez ajouter d'autres attributs bien sûr

}

On peut récupérer AdvertSkills avec findBy :

<?php

// $advert est une instance de Advert

// $advert->getAdvertSkills() n'est pas possible

$listAdvertSkills = $em

->getRepository('OCPlatformBundle:AdvertSkill')

->findBy(array('advert' => $advert))

;

Exemple d’utilisation :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\AdvertSkill;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

// …

public function addAction(Request $request)

{

// On récupère l'EntityManager

$em = $this->getDoctrine()->getManager();

// Création de l'entité Advert

$advert = new *Advert*();

$advert->setTitle('Recherche développeur Symfony.');

$advert->setAuthor('Alexandre');

$advert->setContent("Nous recherchons un développeur Symfony débutant sur Lyon. Blabla…");

// On récupère toutes les compétences possibles

$listSkills = $em->getRepository('OCPlatformBundle:Skill')->findAll();

// Pour chaque compétence

foreach ($listSkills as $skill) {

// On crée une nouvelle « relation entre 1 annonce et 1 compétence »

$advertSkill = new *AdvertSkill*();

// On la lie à l'annonce, qui est ici toujours la même

$advertSkill->setAdvert($advert);

// On la lie à la compétence, qui change ici dans la boucle foreach

$advertSkill->setSkill($skill);

// Arbitrairement, on dit que chaque compétence est requise au niveau 'Expert'

$advertSkill->setLevel('Expert');

// Et bien sûr, on persiste cette entité de relation, propriétaire des deux autres relations

$em->persist($advertSkill);

}

// Doctrine ne connait pas encore l'entité $advert. Si vous n'avez pas défini la relation AdvertSkill

// avec un cascade persist (ce qui est le cas si vous avez utilisé mon code), alors on doit persister $advert

$em->persist($advert);

// On déclenche l'enregistrement

$em->flush();

// … reste de la méthode

}

}

Récupérer une annonce et l’afficher, on met dans view :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

// …

public function viewAction($id)

{

$em = $this->getDoctrine()->getManager();

// On récupère l'annonce $id

$advert = $em

->getRepository('OCPlatformBundle:Advert')

->find($id)

;

if (null === $advert) {

throw new *NotFoundHttpException*("L'annonce d'id ".$id." n'existe pas.");

}

// On avait déjà récupéré la liste des candidatures

$listApplications = $em

->getRepository('OCPlatformBundle:Application')

->findBy(array('advert' => $advert))

;

// On récupère maintenant la liste des AdvertSkill

$listAdvertSkills = $em

->getRepository('OCPlatformBundle:AdvertSkill')

->findBy(array('advert' => $advert))

;

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'advert' => $advert,

'listApplications' => $listApplications,

'listAdvertSkills' => $listAdvertSkills

));

}

}

Et enfin afficher dans avec le Twig :

{# src/OC/PlatformBundle/Resources/view/Advert/view.html.twig #}

{% if listAdvertSkills|length > 0 %}

<div>

Cette annonce requiert les compétences suivantes :

<ul>

{% for advertSkill in listAdvertSkills %}

<li>{{ advertSkill.skill.name }} : niveau {{ advertSkill.level }}</li>

{% endfor %}

</ul>

</div>

{% endif %}

Faites bien la différence entre :

* {{ advertSkill }} qui contient les attributs de la relation, ici le niveau requis via{{ advertSkill.level }} ;
* Et  {{ advertSkill.skill }} qui est la compétence en elle-même (notre entitéSkill), qu'il vous faut utiliser pour afficher le nom de la compétence via{{ advertSkill.skill.name }}.

Attention, dans cet exemple, la méthodefindBy()utilisée dans le contrôleur ne sélectionne que lesAdvertSkill. Donc, lorsque dans la boucle dans la vue on fait{{ advertSkill.skill }}, en réalité Doctrine va effectuer une requête pour récupérer la compétenceSkill associée à cetteAdvertSkill. 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'annonce, cela ferait 20 requêtes : inimaginable.

Une meilleur technique est à venir !

## Les relations bidirectionnelles :

[Comment les créer](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html), et [comment les utiliser.](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/working-with-associations.html)

On va déjà ajouter un attribut à l’entité inverse :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

/\*\*

\* @ORM\Entity

\*/

class Advert

{

/\*\*

\* @ORM\OneToMany(targetEntity="OC\PlatformBundle\Entity\Application", mappedBy="advert")

\*/

private $applications; // Notez le « s », une annonce est liée à plusieurs candidatures

// …

}

Le mappedBy correspond à l’attribut côté propriétaire.

On précise aussi côté propriétaire qu’on est dans une relation bidirectionnelle et non plus unidirectionnelle avec inversedBY, c’est le symétrique de mappedBy.

<?php

// src/OC/PlatformBundle/Entity/Application.php

namespace OC\PlatformBundle\Entity;

/\*\*

\* @ORM\Entity

\*/

class Application

{

/\*\*

\* @ORM\ManyToOne(targetEntity="OC\PlatformBundle\Entity\Advert", inversedBy="applications")

\* @ORM\JoinColumn(nullable=false)

\*/

private $advert;

// …

}

Côté getter et setter on a du coup le paramètre de l’autre côté de la relation :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Table(name="oc\_advert")

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Repository\AdvertRepository")

\*/

class Advert

{

/\*\*

\* @ORM\OneToMany(targetEntity="OC\PlatformBundle\Entity\Application", mappedBy="advert")

\*/

private $applications; // Notez le « s », une annonce est liée à plusieurs candidatures

// … vos autres attributs

public function \_\_construct()

{

$this->applications = new *ArrayCollection*();

// ...

}

public function addApplication(Application $application)

{

$this->applications[] = $application;

}

public function removeApplication(Application $application)

{

$this->applications->removeElement($application);

}

public function getApplications()

{

return $this->applications;

}

// …

}

Maintenant, il faut nous rendre compte d'un petit détail. Voici une petite problématique, lisez bien ce code :

<?php

// Création des entités

$advert = new *Advert*;

$application = new *Application*;

// On lie la candidature à l'annonce

$advert->addApplication($application);

Il ne reste plus qu’à lier l’annonce à la candidature **ET** la candidature à l’annonce !

<?php

// Création des entités

$advert = new *Advert*;

$application = new *Application*;

// On lie la candidature à l'annonce

$advert->addApplication($application);

// On lie l'annonce à la candidature

$application->setAdvert($advert);

Pour ne rien oublier, il vaut mieux les lier, comme ça que l’on fasse l’une ou l’autre, la deuxième sera faite dans la foulée :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

/\*\*

\* @ORM\Entity

\*/

class Advert

{

// …

public function addApplication(Application $application)

{

$this->applications[] = $application;

// On lie l'annonce à la candidature

$application->setAdvert($this);

return $this;

}

public function removeApplication(Application $application)

{

$this->applications->removeElement($application);

// Et si notre relation était facultative (nullable=true, ce qui n'est pas notre cas ici attention) :

// $application->setAdvert(null);

}

// …

}

# Récupérer ses entités avec Doctrine

## Le rôle des repositories :

**Un repositorie centralise tout ce qui touche à la récupération des entités.**

On va construire une liste de méthode qui permet de récupérer par ID, par critère etc …

Ils utilisent directement l’EntityManager.

### Deux façons pour contruire les query et récupérer des entités :

#### Le Doctrine Query Langage (DQL) :

Le DQL est un SQL adapté à Doctrine (aux objets).

SELECT a FROM OCPlatformBundle:Advert a

#### Le Query Builder :

Plus puissant, il crée une requête pas étape.

<?php

$queryBuilder

->select('a')

->from('OCPlatformBundle:Advert', 'a')

;

Imaginons que l’on veuille créer une requête dans lequel on a soit WHERE et il faut écrire AND, sinon on écrit directement WHERE … pas aussi simple que de faire :

<?php

$queryBuilder = ...;

// Je rajoute ma condition, quel que soit ce que j'ai déjà dans mon QueryBuilder :

$queryBuilder->andWhere('champ = 1');

Avec QueryBuilder.

## Les méthodes de récupérations de base :

Les repositories héritent de la classe Doctrine/ORN/EntityRepository

### Find($id) :

Retourne l’entité correspondant à l’ID.

<?php

$repository = $this

->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

;

$advert = $repository->find(5);

// $advert est une instance de OC\PlatformBundle\Entity\Advert

// Correspondant à l'id 5

### findAll() :

Retourne tout ce que contient la BDD.

<?php

$repository = $this

->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

;

$listAdverts = $repository->findAll();

foreach ($listAdverts as $advert) {

// $advert est une instance de Advert

echo $advert->getContent();

}

Ou dans une vue Twig, si l'on a passé la variable $listAdverts au template :

<ul>

{% for advert in listAdverts %}

<li>{{ advert.content }}</li>

{% endfor %}

</ul>

### findBy() :

Plus intéressante, elle permet de selectionner plusieurs entités mais en appliquant un filtre.

<?php

$repository->findBy(

array $criteria,

array $orderBy = null,

$limit = null,

$offset = null

);

Voici un exemple d'utilisation :

<?php

$listAdverts = $repository->findBy(

array('author' => 'Alexandre'), // Critere

array('date' => 'desc'), // Tri

5, // Limite

0 // Offset

);

foreach ($listAdverts as $advert) {

// $advert est une instance de Advert

echo $advert->getContent();

}

Ici on récupère les entités ayant comme auteur « Alexandre », par ordre descendant, on prend les 5 premiers à partir de 0.

### findOneBy() :

Fonctionne comme findBy sauf qu’il en renvoit un, donc on n’a pas la limite.

<?php

$advert = $repository->findOneBy(array('author' => 'Marine'));

// $advert est une instance de Advert

S’il y en a plusieurs, c’est le premier qui est renvoyé.

### Les méthodes magiques : findByX et findOneByX :

**findByX($valeur) :**

Fonctionne différament en fonction de l’entité sur lequel on l’applique. Avec Advert, on a findByAuthor, findByTitle …

<?php

$listAdverts = $repository->findByAuthor('Alexandre');

// $listAdverts est un Array qui contient toutes les annonces

// écrites par Alexandre

On a donc findBy(array('author' => 'Alexandre'))  qui est strictement égal à findByAuthor('Alexandre') .

**findOneByX :**

Même principe, dépent de l’entité sur lequel on l’applique : ici finOneByAuthor …

<?php

$advert = $repository->findOneByTitle('Recherche développeur.');

// $advert est une instance d'Advert dont le titre

// est "Recherche développeur." ou null si elle n'existe pas.

## Méthodes de récupération personnelle :

Pour développer nos propres méthodes, on distingue trois types :

* QueryBuilder
* LaQuery
* Les résultats

### Le QueryBuilder :

Pour récupérer le QueryBuilder, on utilise l’EntityManager. Il dispose d’une méthode **createQueryBuilder()**, qui retourne une instance de QueryBuilder. L’EntityManager est disponible depuis un respository en utilisant \_em au lieu de em. Soit : $this->\_em.

**Le code complet pour récupérer un QueyBuilder neuf depuis le repository : $this->\_em->createQueryBuilder().**

[Méthode du createQueryBuilder.](https://github.com/doctrine/doctrine2/blob/master/lib/Doctrine/ORM/EntityRepository.php#L80)

Le repository dispose d’une méthode createQueryBuilder($alias) qui utilise la méthode de l’EntityManager mais en définissant pour nous le SELECT et le FROM.

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'annonce cela serait donc un « a ».

<?php

// src/OC/PlatformBundle/Entity/AdvertRepository.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\EntityRepository;

class AdvertRepository extends EntityRepository

{

public function myFindAll()

{

// Méthode 1 : en passant par l'EntityManager

$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 OC\PlatformBundle\Entity\Advert

// Méthode 2 : en passant par le raccourci (je recommande)

$queryBuilder = $this->createQueryBuilder('a');

// On n'ajoute pas de critère ou tri particulier, la construction

// de notre requête est finie

// On récupère la Query à partir du QueryBuilder

$query = $queryBuilder->getQuery();

// On récupère les résultats à partir de la Query

$results = $query->getResult();

// On retourne ces résultats

return $results;

}

}

myFindAll() retourne exactement le même résultat que findAll() càd un tableau de toutes les entités Advert dans la BDD :

Voici la méthode sans les commentaires :

<?php

public function myFindAll()

{

return $this

->createQueryBuilder('a')

->getQuery()

->getResult()

;

}

Depuis un controller, on fait comme pour les autres méthodes :

<?php

// Depuis un contrôleur

public function testAction()

{

$repository = $this

->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

;

$listAdverts = $repository->myFindAll();

// ...

}

Pour le moment on récupère juste mais on n’applique rien. Ici une méthode équivalente au find($id).

<?php

// Dans un repository

public function myFindOne($id)

{

$qb = $this->createQueryBuilder('a');

$qb

->where('a.id = :id')

->setParameter('id', $id)

;

return $qb

->getQuery()

->getResult()

;

}

<?php

// Depuis un repository

public function findByAuthorAndDate($author, $year)

{

$qb = $this->createQueryBuilder('a');

$qb->where('a.author = :author')

->setParameter('author', $author)

->andWhere('a.date < :year')

->setParameter('year', $year)

->orderBy('a.date', 'DESC')

;

return $qb

->getQuery()

->getResult()

;

}

Exemple : Annonce postée durant l’année en cours :

<?php

// src/OC/PlatformBundle/Entity/AdvertRepository.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\EntityRepository;

// N'oubliez pas ce use

use Doctrine\ORM\QueryBuilder;

class AdvertRepository extends EntityRepository

{

public function whereCurrentYear(QueryBuilder $qb)

{

$qb

->andWhere('a.date BETWEEN :start AND :end')

->setParameter('start', new \Datetime(date('Y').'-01-01')) // Date entre le 1er janvier de cette année

->setParameter('end', new \Datetime(date('Y').'-12-31')) // Et le 31 décembre de cette année

;

}

}

Ici on en traite pas un Query mais bien un QueryBuilder.

Pour l’utiliser on l’imbrique dans cette méthode :

<?php

// Depuis un repository

public function myFind()

{

$qb = $this->createQueryBuilder('a');

// On peut ajouter ce qu'on veut avant

$qb

->where('a.author = :author')

->setParameter('author', 'Marine')

;

// On applique notre condition sur le QueryBuilder

$this->whereCurrentYear($qb);

// On peut ajouter ce qu'on veut après

$qb->orderBy('a.date', 'DESC');

return $qb

->getQuery()

->getResult()

;

}

[Toutes les autres méthodes de QueryBuilder.](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html)

## Méthode La Query :

Sert surtout à la gestion du cache des requêtes.

### GetResult() :

Restourne un tableau même s’il n’y a qu’un résultat.

<?php

$listAdverts = $qb->getQuery()->getResult();

foreach ($listAdverts as $advert) {

// $advert est une instance d'Advert dans notre exemple

$advert->getContent();

}

### GetArrayResult() :

Renvoie aussi sous forme de tableau même s’il n’y a qu’une réponse. Renvoie un tableau de valeur et non d’objet, pratique quant on ne veut **QUE LIRE** et non faire des modifications 🡺 plus rapide.

<?php

$listAdverts = $qb->getQuery()->getArrayResult();

foreach ($listAdverts as $advert) {

// $advert est un tableau

// Faire $advert->getContent() est impossible. Vous devez faire :

$advert['content'];

}

Heureusement, Twig est intelligent :{{ advert.content }}exécute$advert->getContent()si$advert est un objet, et exécute$advert['content']sinon. Du point de vue de Twig, vous pouvez utilisergetResult()ougetArrayResult()indifféremment.

Comme on ne veut que lire avec cette méthode, rien n’est enregistré dans la BDD au prochain flush().

### GetOneOrNullResult() :

Exécute la requête et retourne un résultat ou NULL. Elle retourne donc une instance de l’entité comme getResult().

Si plusieurs résultats sont renvoyés, retourne une erreur.

### getSingleResult() :

Comme getOneOrNullResult() sauf qu’elle renvoie une erreur si aucun résultat.

### Execute() :

Execute la requête, principalement utilisé pour executer des requêtes quine retournent pas de résultat comme UPDATE, INSET INTO …

Cependant, toutes les autres méthodes que nous venons de voir ne sont en fait que des raccourcis vers cette méthodeexecute(), en changeant juste le mode d'hydratation des résultats (objet, tableau, etc.).

<?php

// Voici deux méthodes strictement équivalentes :

$results = $query->getArrayResult();

// Et :

$results = $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

### Utilisation du DQL :

[Le DQL est bien documenté](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html).

**Pour créer une requête on utilise createQuery() de l’EntityManager.**

<?php

// Depuis un repository

public function myFindAllDQL()

{

$query = $this->\_em->createQuery('SELECT a FROM OCPlatformBundle:Advert a');

$results = $query->getResult();

return $results;

}

La requête :

SELECT a FROM OCPlatformBundle:Advert a

On sélectionne bien un objet, comme annoncé depuis le début, c’est équivalent au « \* » de SQL, ici on sélectionne tous les attributs de l’entité. On peut toujours sélectionner certains attributs, comme a.title, mais on aurait un tableau en retour et non un objet 🡺 pas de modification au flush.

[Documentation du DQL](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html)

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 commandedoctrine: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 bin/console doctrine:query:dql "SELECT a FROM OCPlatformBundle:Advert a".

##### Exemples :

Pour faire une jointure :

SELECT a, u FROM Advert a JOIN a.user u WHERE u.age = 25

Pour utiliser une fonction SQL (attention toutes les fonctions SQL ne sont pas implémentées en DQL) :

SELECT a FROM Advert a WHERE TRIM(a.author) = 'Alexandre'

Pour sélectionner seulement un attribut (attention les résultats seront donc sous forme de tableaux et non d'objets) :

SELECT a.title FROM Advert a WHERE a.id IN(1, 3, 5)

Et bien sûr vous pouvez également utiliser des paramètres :

<?php

public function myFindDQL($id)

{

$query = $this->\_em->createQuery('SELECT a FROM Advert a WHERE a.id = :id');

$query->setParameter('id', $id);

// Utilisation de getSingleResult car la requête ne doit retourner qu'un seul résultat

return $query->getSingleResult();

}

## Utiliser les jointures dans les requêtes :

### Pourquoi utiliser les jointures ?

On fait des jointures quand on veut récupérer des entiés B liés à l’entité A. Pour éviter de devoir boucler sur toutes les liaisons de toutes les entités, (ce qui fairait énormément de requêtes), on joint.

### Jointures avec le Query Builder :

<?php

// Depuis le repository d'Advert

public function getAdvertWithApplications()

{

$qb = $this

->createQueryBuilder('a')

->leftJoin('a.applications', 'app')

->addSelect('app')

;

return $qb

->getQuery()

->getResult()

;

}

Rien d'extravagant donc ! Voici l'idée un peu plus en détails :

* D'abord on crée une jointure avec la méthodeleftJoin() (ou  innerJoin() pour faire l'équivalent d'unINNER JOIN). Le premier argument de la méthode est l'attribut de l'entité principale (celle qui est dans leFROM de la requête) sur lequel faire la jointure. Dans l'exemple, l'entitéAdvert possède un attributapplications. Le deuxième argument de la méthode est l'alias de l'entité jointe (arbitraire).
* Puis on sélectionne également l'entité jointe, via unaddSelect(). En effet, unselect('app') tout court aurait écrasé leselect('a') déjà fait par le ‌createQueryBuilder(), rappelez-vous.

Attention : on ne peut faire une jointure que si l'entité duFROMpossède un attribut vers l'entité à joindre ! Cela veut dire que soit l'entité duFROMest l'entité propriétaire de la relation, soit la relation est bidirectionnelle.  
Dans notre exemple, la relation entreAdvert etApplication est une *Many-To-One*avecApplication du 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 attributapplications dans l'entité inverseAdvert. C'est ce que nous avons fait à la fin du chapitre précédent.

*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 unONhabituellement 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 ! Il est donc inutile de lui préciser leON.

Bien sûr, vous pouvez toujours personnaliser la condition de jointure, en *rajoutant* vos conditions à la suite duONgénéré par Doctrine, grâce à la syntaxe duWITH:

<?php

$qb->innerJoin('a.applications', 'app', 'WITH', 'YEAR(app.date) > 2013')

Le troisième argument est le type de conditionWITH, et le quatrième argument est ladite condition.

En SQL, la différence entre leONet leWITHest simple : unON*définit* la condition pour la jointure, alors qu'unWITH*ajoute* une condition pour la jointure. Attention, en DQL leONn'existe pas, seul leWITHest supporté. Ainsi, la syntaxe précédente avec leWITHserait équivalente à la syntaxe SQL suivante à base deON:

SELECT \*

FROM Advert a

INNER JOIN Application app ON (app.advert\_id = a.id AND YEAR(app.*date*) > 2013)

Grâce auWITH, on n'a pas besoin de réécrire la condition par défaut de la jointure, le app.advert\_id = a.id.

On pourrait utiliser les résultats comme ceci :

<?php

// Depuis un contrôleur

public function listAction()

{

$listAdverts = $this

->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

->getAdvertWithApplications()

;

foreach ($listAdverts as $advert) {

// Ne déclenche pas de requête : les candidatures sont déjà chargées !

// Vous pourriez faire une boucle dessus pour les afficher toutes

$advert->getApplications();

}

// …

}

# Evènements et extensions Doctrine

## Les évènements :

Ce sont des actions intervenant juste avant, après, pendant une autre action. Exemple, si on modifie une annonce on doit modifier aussi la date. Pour éviter de se répeter ou de recopier du code, on fait référence aux **CALBACKS du Lifcycle**.

On va rajouter un attribut à l’entité Advert :

<?php

/\*\*

\* @ORM\Column(name="updated\_at", type="datetime", nullable=true)

\*/

private $updatedAt;

Ensutie on rpévient Doctrine que l’entité contient des callbacks :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

\* @ORM\HasLifecycleCallbacks()

\*/

class Advert

{

// …

}

Avec HasLifeCycleCallbacks(). Si on n’écrit pas ça les callbacks seront ignorés.

On va ensuite créer une méthode pour mettre à jour automatiquement la date :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

/\*\*

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

\* @ORM\HasLifecycleCallbacks()

\*/

class Advert

{

// …

public function updateDate()

{

$this->setUpdatedAt(new \Datetime());

}

}

Maintenant on va dire qu’il faut écouter l’évènement, et executer la méthode **AVANT** que l’entité soit modifié dans la BDD :

<?php

/\*\*

\* @ORM\PreUpdate

\*/

public function updateDate()

On ne met aucun argument dans un évènement, on ne peut modifier que l’élément courant. Pour faire des choses plus complexex 🡺 services.

|  |  |
| --- | --- |
| **Évènement** | **Description** |
| PrePersist | L'évènement PrePersist se produit juste avant que l'EntityManager ne persiste effectivement l'entité. Concrètement, cela exécute le callback juste avant un $em->persist($entity). 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 postPersist se produit juste après que l'EntityManager ait effectivement persisté l'entité. Attention, cela n'exécute pas le callback juste après le $em->persist($entity), mais juste après le $em->flush(). À l'inverse du prePersist, 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 flush(). |
| PreUpdate | L'évènement preUpdate 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 $em->flush(). Le callback sera exécuté juste avant le flush(). 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 postUpdate 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. |
| PreRemove | L'évènement PreRemove se produit juste avant que l'EntityManager ne supprime une entité, c'est-à-dire juste avant un $em->flush() qui précède un $em->remove($entite). 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 flush(). |
| PostRemove | L'évènement PostRemove 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. |
| PostLoad | L'évènement PostLoad se produit juste après que l'EntityManager a chargé une entité (ou après un $em->refresh()). Utile pour appliquer une action lors du chargement d'une entité. |

**Autre exemple :**

On veut un compteur de candidature pour les annonces. On pourrait affecter un COUNT(\*) pour être dynamique, ou alors faire un évènement qui fait +1 à chaque nouvelle candidature et -1 à chaque suppression :

On ne peut pas utiliser **POSTUPDATE** car il serait mis à jour après la mise à jour de la BDD, du coup il serait trop tard pour enregistrer notre attribut.

Pour le reste, ça fonctionne de la même manière :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

class Advert

{

/\*\*

\* @ORM\Column(name="nb\_applications", type="integer")

\*/

private $nbApplications = 0;

public function increaseApplication()

{

$this->nbApplications++;

}

public function decreaseApplication()

{

$this->nbApplications--;

}

// ...

}

Ensuite, on doit définir deux callbacks dans l'entité Application pour mettre à jour le compteur de l'entité Advert liée. Notez bien que nos évènements concernent bien l'entité Application, et non l'entité Advert ! Voici comment on pourrait faire :

<?php

// src/OC/PlatformBundle/Entity/Application.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Table(name="oc\_application")

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\ApplicationRepository")

\* @ORM\HasLifecycleCallbacks()

\*/

class Application

{

/\*\*

\* @ORM\PrePersist Juste avant le persiste

\*/

public function increase()

{

$this->getAdvert()->increaseApplication();

}

/\*\*

\* @ORM\PreRemove Juste avant le Remove

\*/

public function decrease()

{

$this->getAdvert()->decreaseApplication();

}

// ...

}

### Utiliser des services :

Les services sont executés dans un callbacks contenant toutes nos entités, et non attaché à un seul évènement.

A chaque fois qu’une candidature est reçu on envoit un mail. Se fait en deux parties :

* L’envoi d’un e-mail
* L’aspect systématique à chaque candidature

On les sépare parce qu’on pourra utiliser l’envoi d’email pour autre chose plus tard.

**L’envoi de l’e-mail :**

<?php

// src/OC/PlatformBundle/Email/ApplicationMailer.php

namespace OC\PlatformBundle\Email;

use OC\PlatformBundle\Entity\Application;

class ApplicationMailer

{

/\*\*

\* @var \Swift\_Mailer

\*/

private $mailer;

public function \_\_construct(\Swift\_Mailer $mailer)

{

$this->mailer = $mailer;

}

public function sendNewNotification(Application $application)

{

$message = new \Swift\_Message(

'Nouvelle candidature',

'Vous avez reçu une nouvelle candidature.'

);

$message

->addTo($application->getAdvert()->getAuthor()) // Ici bien sûr il faudrait un attribut "email", j'utilise "author" à la place

->addFrom('admin@votresite.com')

;

$this->mailer->send($message);

}

}

Ainsi que sa configuration :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.email.application\_mailer:

class: OC\PlatformBundle\Email\ApplicationMailer

arguments:

- "@mailer"

**La partie callback :**

<?php

// src/OC/PlatformBundle/DoctrineListener/ApplicationCreationListener.php

namespace OC\PlatformBundle\DoctrineListener;

use Doctrine\Common\Persistence\Event\LifecycleEventArgs;

use OC\PlatformBundle\Email\ApplicationMailer;

use OC\PlatformBundle\Entity\Application;

class ApplicationCreationListener

{

/\*\*

\* @var ApplicationMailer

\*/

private $applicationMailer;

public function \_\_construct(ApplicationMailer $applicationMailer)

{

$this->applicationMailer = $applicationMailer;

}

public function postPersist(LifecycleEventArgs $args)

{

$entity = $args->getObject();

// On ne veut envoyer un email que pour les entités Application

if (!$entity instanceof Application) {

return;

}

$this->applicationMailer->sendNewNotification($entity);

}

}

La méthode doit respecter le même nom que l’évènement :

* Le seul argument donné à la méthode est LifecycleEventArgs qui offre deux méthodes : getObject et getObjectManager.
* On vérifie ensuite qu’on a bien affaire à une application et pas autre chose avec instanceof.

Enfin :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.doctrine\_listener.application\_creation:

class: OC\PlatformBundle\DoctrineListener\ApplicationCreationListener

arguments:

- "@oc\_platform.email.application\_mailer"

tags:

- { name: doctrine.event\_listener, event: postPersist }

Notez également qu'à la fin de la transaction, Doctrine fait unROLLBACK et non unCOMMIT. Cela signifie qu'il annule toute la transaction, et donc l'enregistrement de notre annonce et de nos candidatures. Annuler toute la transaction parce qu'un e-mail de notification n'a pas pu être‌ envoyé, ce n'est généralement pas le comportement souhaité. Pour éviter cela il faudrait, dans leApplicationCreationListener, rajouter un bloc [try/catch](https://php.net/manual/en/language.exceptions.php) afin d'intercepter l'exception de SwiftMailer, et ainsi ne pas annuler toute la transaction Doctrine. Pensez-y !

## Les extensions Doctrine :

Il y a des comportements communs que l’on veut implémenter. Il existe des extensions pour éviter de se répeter sans cesse. C’est utiliser la philosophie **DRY**.

### Installer le SoftDoctrineExtensionBundle :

Il permet d’intégrer des extensions au projet Symfony.

// composer.json

"require": {

"stof/doctrine-extensions-bundle": "^1.2.2"

}

Ce bundle intègre la bibliothèque [DoctrineExtensions](https://github.com/Atlantic18/DoctrineExtensions) sous-jacente, qui est celle qui inclut réellement les extensions Doctrine.

N'oubliez pas d'enregistrer le bundle dans le noyau :

<?php

// app/AppKernel.php

public function registerBundles()

{

return array(

// …

new *Stof*\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),

// …

);

}

**Utiliser une extension : l’exemple de Sluggable :**

Il permet de définir un attribut **SLUG** à l’entité.

Il faut d’abord l’activer avec config.yml :

# app/config/config.yml

# Stof\DoctrineExtensionsBundle configuration

stof\_doctrine\_extensions:

orm:

default:

sluggable: true

# Créer des formulaires

## Gestion des formulaires :

### Qu’est ce que c’est ?

**Un formulaire se construit sur un objet existant et son objectif est d’hydrater cet objet.**

On parle bien d’objet et non d’entité Doctrine.

Rappel de la classe Advert sans les annotations :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

class Advert

{

private $id;

private $date;

private $title;

private $author;

private $content;

private $published = true;

private $image;

private $categories;

private $applications;

private $updatedAt;

private $nbApplications = 0;

private $slug;

public function \_\_construct()

{

$this->date = new \Datetime();

$this->categories = new *ArrayCollection*();

$this->applications = new *ArrayCollection*();

}

// … Les getters et setters

}

**Les getters et setters sont importants, si on avait eu set\_Title, ça n’aurait pas marché afec Form. De plus le formulaire ne sert QUE à hydrater. Une fois que l’on a hydrater, on peut faire le reste (BDD, requête …).**

On dit **hydrater**, c’est-à-dire : $Advert->setAuthor(‘Alexandre’) ;

On va modifier addAction :

Pour créer un formulaire il faut :

* Un objet
* Un constructeur de formulaire : **formBuilder**

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\Form\Extension\Core\Type\CheckboxType;

use Symfony\Component\Form\Extension\Core\Type\DateType;

use Symfony\Component\Form\Extension\Core\Type\FormType;

use Symfony\Component\Form\Extension\Core\Type\SubmitType;

use Symfony\Component\Form\Extension\Core\Type\TextareaType;

use Symfony\Component\Form\Extension\Core\Type\TextType;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// On crée un objet Advert

$advert = new *Advert*();

// On crée le FormBuilder grâce au service form factory

$formBuilder = $this->get('form.factory')->createBuilder(*FormType*::class, $advert);

// On ajoute les champs de l'entité que l'on veut à notre formulaire

$formBuilder

->add('date', *DateType*::class)

->add('title', *TextType*::class)

->add('content', *TextareaType*::class)

->add('author', *TextType*::class)

->add('published', *CheckboxType*::class)

->add('save', *SubmitType*::class)

;

// Pour l'instant, pas de candidatures, 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('OCPlatformBundle:Advert:add.html.twig', array(

'form' => $form->createView(),

));

}

}

Par exemple, au lieu de TextType::class, nous aurions pu mettre'Symfony\Component\Form\Extension\Core\Type\TextType', les deux sont strictement équivalents.

Le bouton SAVE n’est pas mappé avec le formulaire, on pourrait le faire à la main plus tard.

**Pour générer le formulaire :**

{# src/OC/PlatformBundle/Resources/views/Advert/form.html.twig #}

<h3>Formulaire d'annonce</h3>

<div class="well">

{{ form(form) }}

</div>

On ajoute ça dans form puisque add contient form.

Certains champs sont pré-remplis, c’est à cause de notre entité, par exemple, published est à truze par défaut.

### Ajouter des champs :

On respecte la sémantique observée : nom, type, option.

| **Texte** | **Choix** | **Date et temps** | **Divers** | **Multiple** | **Caché** |
| --- | --- | --- | --- | --- | --- |
| [TextType](https://symfony.com/doc/current/reference/forms/types/text.html) [TextareaType](https://symfony.com/doc/current/reference/forms/types/textarea.html) [EmailType](https://symfony.com/doc/current/reference/forms/types/email.html) [IntegerType](https://symfony.com/doc/current/reference/forms/types/integer.html) [MoneyType](https://symfony.com/doc/current/reference/forms/types/money.html) [NumberType](https://symfony.com/doc/current/reference/forms/types/number.html) [PasswordType](https://symfony.com/doc/current/reference/forms/types/password.html) [PercentType](https://symfony.com/doc/current/reference/forms/types/percent.html) [SearchType](https://symfony.com/doc/current/reference/forms/types/search.html) [UrlType](https://symfony.com/doc/current/reference/forms/types/url.html)  [RangeType](https://symfony.com/doc/current/reference/forms/types/range.html) | [ChoiceType](https://symfony.com/doc/current/reference/forms/types/choice.html) [EntityType](https://symfony.com/doc/current/reference/forms/types/entity.html) [CountryType](https://symfony.com/doc/current/reference/forms/types/country.html) [LanguageType](https://symfony.com/doc/current/reference/forms/types/language.html) [LocaleType](https://symfony.com/doc/current/reference/forms/types/locale.html) [TimezoneType](https://symfony.com/doc/current/reference/forms/types/timezone.html)  [CurrencyType](https://symfony.com/doc/current/reference/forms/types/currency.html) | [DateType](https://symfony.com/doc/current/reference/forms/types/date.html) [DatetimeType](https://symfony.com/doc/current/reference/forms/types/datetime.html) [TimeType](https://symfony.com/doc/current/reference/forms/types/time.html) [BirthdayType](https://symfony.com/doc/current/reference/forms/types/birthday.html) | [CheckboxType](https://symfony.com/doc/current/reference/forms/types/checkbox.html) [FileType](https://symfony.com/doc/current/reference/forms/types/file.html) [RadioType](https://symfony.com/doc/current/reference/forms/types/radio.html) | [CollectionType](https://symfony.com/doc/current/reference/forms/types/collection.html) [RepeatedType](https://symfony.com/doc/current/reference/forms/types/repeated.html) | [HiddenType](https://symfony.com/doc/current/reference/forms/types/hidden.html) [CsrfType](https://symfony.com/doc/current/reference/forms/types/csrf.html) |

Il faut toujours faire correspondre le type dans le formulaire avec son type d’origine. Si on met texte et boolean ensemble ça ne collera pas.

Exemple pour une entité Doctrine :

<?php

/\*\*

\* @ORM\Column(name="published", type="boolean")

\*/

private $published = true;

Il nous faut donc un type de champ qui retourne un *boolean*, à savoir CheckboxType :

<?php

$formBuilder->add('published', *CheckboxType*::class);

On va d’abords vérifier que la requête est de type POST (donc qu’on a clické sur submit). Ensuite avec la méthode handleRequest, avec laquelle on va lire le formulaire, récupérer les valeurs, on hydrate les objets. Seulement après on peut les traiter.

On fait un isValid() pour vérifier que les valeurs correspondent au type de valeur attendu :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\Form\Extension\Core\Type\CheckboxType;

use Symfony\Component\Form\Extension\Core\Type\DateType;

use Symfony\Component\Form\Extension\Core\Type\FormType;

use Symfony\Component\Form\Extension\Core\Type\SubmitType;

use Symfony\Component\Form\Extension\Core\Type\TextareaType;

use Symfony\Component\Form\Extension\Core\Type\TextType;

use Symfony\Component\HttpFoundation\Request;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// On crée un objet Advert

$advert = new *Advert*();

// J'ai raccourci cette partie, car c'est plus rapide à écrire !

$form = $this->get('form.factory')->createBuilder(*FormType*::class, $advert)

->add('date', *DateType*::class)

->add('title', *TextType*::class)

->add('content', *TextareaType*::class)

->add('author', *TextType*::class)

->add('published', *CheckboxType*::class, array('required' => false))

->add('save', *SubmitType*::class)

->getForm()

;

// Si la requête est en POST

if ($request->isMethod('POST')) {

// On fait le lien Requête <-> Formulaire

// À partir de maintenant, la variable $advert contient les valeurs entrées dans le formulaire par le visiteur

$form->handleRequest($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 $advert dans la base de données, par exemple

$em = $this->getDoctrine()->getManager();

$em->persist($advert);

$em->flush();

$request->getSession()->getFlashBag()->add('notice', 'Annonce bien enregistrée.');

// On redirige vers la page de visualisation de l'annonce nouvellement créée

return $this->redirectToRoute('oc\_platform\_view', array('id' => $advert->getId()));

}

}

// À ce stade, le formulaire n'est pas valide car :

// - 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 contient des valeurs invalides, donc on l'affiche de nouveau

return $this->render('OCPlatformBundle:Advert:add.html.twig', array(

'form' => $form->createView(),

));

}

}

Grace à isValid(), l’annonce ne sera enregistré que si tout va bien.

Ici ça ne marche que si published est coché, sauf qu’on veut avoir le choix ! C’est là qu’intervient les options (sous forme de tableau).

<?php

$formBuilder->add('published', *CheckboxType*::class, array('required' => false))

Un message apparaît si on décoche published.

Pourtant, nous n'avons pas utilisé de JavaScript ! C'est juste du HTML5. En mettant l'attributrequired="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 HMTL5, il pourra valider le formulaire sans problème.

### Gérer les valeurs par défaut :

**En modifiant l’entité avant de l’ajouter au formulaire, on aura une valeur par défaut.**

<?php

// On crée une nouvelle annonce

$advert = new *Advert*;

// 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

$advert->setDate(new \Datetime());

// Et on construit le formBuilder avec cette instance d'annonce

$formBuilder = $this->get('form.factory')->createBuilder(*FormType*::class, $advert);

// N'oubliez pas d'ajouter les champs comme précédemment avec la méthode ->add()

Et si on veut modifier une annonce déjà enregistrée en base de données, alors il suffit de la récupérer avant la création du formulaire, comme ceci :

<?php

// Récupération d'une annonce déjà existante, d'id $id.

$advert = $this->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

->find($id)

;

// Et on construit le formBuilder avec cette instance de l'annonce, comme précédemment

$formBuilder = $this->get('form.factory')->createBuilder(*FormType*::class, $advert);

// N'oubliez pas d'ajouter les champs comme précédemment avec la méthode ->add()

### Personnaliser l’affichage du formulaire :

On peut utiliser Boostrap par défaut dans Twig :

# app/config/config.yml

twig:

form\_themes:

- 'bootstrap\_3\_layout.html.twig'

Pour avoir plus de flexibilité :

{# src/OC/PlatformBundle/Resources/views/Advert/form.html.twig #}

<h3>Formulaire d'annonce</h3>

<div class="well">

{{ form\_start(form, {'attr': {'class': 'form-horizontal'}}) }}

{# Les erreurs générales du formulaire. #}

{{ form\_errors(form) }}

{# Génération du label + error + widget pour un champ. #}

{{ form\_row(form.date) }}

{# Génération manuelle et éclatée : #}

<div class="form-group">

{# Génération du label. #}

{{ form\_label(form.title, "Titre de l'annonce", {'label\_attr': {'class': 'col-sm-2 control-label'}}) }}

{# Affichage des erreurs pour ce champ précis. #}

{{ form\_errors(form.title) }}

<div class="col-sm-10">

{# Génération de l'input. #}

{{ form\_widget(form.title, {'attr': {'class': 'form-control'}}) }}

</div>

</div>

{# Idem pour un autre champ. #}

<div class="form-group">

{{ form\_label(form.content, "Contenu de l'annonce", {'label\_attr': {'class': 'col-sm-2 control-label'}}) }}

{{ form\_errors(form.title) }}

<div class="col-sm-10">

{{ form\_widget(form.content, {'attr': {'class': 'form-control'}}) }}

</div>

</div>

{{ form\_row(form.author) }}

{{ form\_row(form.published) }}

{# Pour le bouton, pas de label ni d'erreur, on affiche juste le widget #}

{{ form\_widget(form.save, {'attr': {'class': 'btn btn-primary'}}) }}

{# Génération automatique des champs pas encore écrits.

Dans cet exemple, ce serait le champ CSRF (géré automatiquement par Symfony !)

et tous les champs cachés (type « hidden »). #}

{{ form\_rest(form) }}

{# Fermeture de la balise <form> du formulaire HTML #}

{{ form\_end(form) }}

</div>

Revenons rapidement sur les fonctions Twig que j'ai utilisées :

* form\_start() affiche la balise d'ouverture du formulaire HTML, soit <form>. Il faut passer la variable du formulaire en premier argument, et les paramètres en deuxième argument. L'index attr des paramètres, et cela s'appliquera à toutes les fonctions suivantes, représente les attributs à ajouter à la balise HTML générée, ici le<form>. Il nous permet d'appliquer une classe CSS au formulaire, ici form-horizontal.
* form\_errors() affiche les erreurs attachées au champ donné en argument. Nous verrons les erreurs de validation dans le prochain chapitre.
* form\_label() affiche le label HTML du champ donné en argument. Le deuxième argument est le contenu du label.
* form\_widget() affiche le champ HTML lui-même (que ce soit<input>,<select>, etc.).
* form\_row() affiche le label, les erreurs et le champ en même temps, en respectant la vue définit dans le thème du formulaire que vous utilisez.
* form\_rest() affiche tous les champs manquants du formulaire (dans notre cas, juste le champ CSRF puisque nous avons déjà affiché à la main tous les autres champs).
* form\_end() affiche la balise de fermeture du formulaire HTML, soit</form>.

L'habillage des formulaires est un sujet complexe : personnalisation d'un champ en particulier, de tous les champs d'un même type, etc. Toutes les fonctions Twig que nous avons vu sont également personnalisable. Je vous invite vivement à consulter la [documentation à ce sujet](https://symfony.com/doc/current/book/forms.html#form-theming) qui vous permettra d'aller beaucoup plus loin. Cela s'appelle en anglais le form theming.

Le **CSFR : Cross Site Request Forgeries** permet de vérifier que celui qui valide le formulaire est le même que celui qui l’a affiché.

On peut personnaliser le type du champ :

Je ne vais pas décrire la démarche ici, mais sachez que cela existe et que [la documentation traite ce point](https://symfony.com/doc/current/cookbook/form/create_custom_field_type.html).

## Externaliser la définition de ses formulaires :

Pour que le formulaire fonctionne aussi ailleurs que dans le contrôleur, il faut donc le détacher dans une classe à part, nommée AdvertType par convention.

On va mettre notre xxxTpe.php dans form du Bundle.

php bin/console doctrine:generate:form OCPlatformBundle:Advert

On modifie le fichier pour qu’il corresponde à ce qu’on a crée avant :

<?php

namespace OC\PlatformBundle\Form;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\Extension\Core\Type\CheckboxType;

use Symfony\Component\Form\Extension\Core\Type\DateTimeType;

use Symfony\Component\Form\Extension\Core\Type\SubmitType;

use Symfony\Component\Form\Extension\Core\Type\TextareaType;

use Symfony\Component\Form\Extension\Core\Type\TextType;

use Symfony\Component\Form\FormBuilderInterface;

use Symfony\Component\OptionsResolver\OptionsResolver;

class AdvertType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('date', *DateTimeType*::class)

->add('title', *TextType*::class)

->add('author', *TextType*::class)

->add('content', *TextareaType*::class)

->add('published', *CheckboxType*::class, array('required' => false))

->add('save', *SubmitType*::class);

}

public function configureOptions(OptionsResolver $resolver)

{

$resolver->setDefaults(array(

'data\_class' => 'OC\PlatformBundle\Entity\Advert'

));

}

}

On sait qu’un formulaire se construit avec un objet, on l’a défini avec data\_class, pour dire que l’on parle bien d’Advert.

Maintenant, côté contrôleur, on crée un formulaire avec **CREATE()** du service form.factory et non plus createBuilder().

<?php

// Dans le contrôleur

$advert = new *Advert*;

$form = $this->get('form.factory')->create(*AdvertType*::class, $advert);

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;

use OC\PlatformBundle\Form\AdvertType;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

$advert = new *Advert*();

$form = $this->get('form.factory')->create(*AdvertType*::class, $advert);

if ($request->isMethod('POST') && $form->handleRequest($request)->isValid()) {

$em = $this->getDoctrine()->getManager();

$em->persist($advert);

$em->flush();

$request->getSession()->getFlashBag()->add('notice', 'Annonce bien enregistrée.');

return $this->redirectToRoute('oc\_platform\_view', array('id' => $advert->getId()));

}

return $this->render('OCPlatformBundle:Advert:add.html.twig', array(

'form' => $form->createView(),

));

}

}

Vous trouvez que faire $form = $this->get('form.factory')->create(AdvertType::class, $advert) est trop long ? Rien que pour vous, le contrôleur de base que nous héritons dispose d'une méthode raccourcie, voici l'équivalent : $form = $this->createForm(AdvertType::class, $advert).

## Les formulaires imbriqués :

On souhaite par exemple mettre une application sur un Advert, mais on veut mettre une adresse à notre application en même temps. 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 :

php bin/console doctrine:generate:form OCPlatformBundle:Image

En explicitant les types des champs, cela donne le code suivant :

<?php

// src/OC/PlatformBundle/Form/ImageType.php

namespace OC\PlatformBundle\Form;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\Extension\Core\Type\TextType;

use Symfony\Component\Form\FormBuilderInterface;

use Symfony\Component\OptionsResolver\OptionsResolver;

class ImageType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('url', *TextType*::class)

->add('alt', *TextType*::class);

}

public function configureOptions(OptionsResolver $resolver)

{

$resolver->setDefaults(array(

'data\_class' => 'OC\PlatformBundle\Entity\Image'

));

}

}

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 Advert 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 :

Il faut pour celà, qu’il y ait une entité One-To-One ou Many-To-One.

Ici on imbrique image dans Advert :

<?php

// src/OC/PlatformBundle/Form/AdvertType.php

class AdvertType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('date', *DateTimeType*::class)

->add('title', *TextType*::class)

->add('author', *TextType*::class)

->add('content', *TextareaType*::class)

->add('published', *CheckboxType*::class, array('required' => false))

->add('image', *ImageType*::class) // Ajoutez cette ligne

->add('save', *SubmitType*::class);

}

// ...

}

Ils sont affichés après le bouton save, parce qu’on ne les a pas affichés à la main.

Je fais un petit apparté Doctrine sur une erreur qui arrive souvent. Si jamais lorsque vous validez votre formulaire vous avez une erreur de ce type :

A new entity was found through the relationship 'OC\PlatformBundle\Entity\Advert#image'

that was not configured to cascade persist operations for entity:

OC\PlatformBundle\Entity\Image@000000000579b29e0000000061a76c55. To solve this issue:

Either explicitly call EntityManager#persist() on this unknown entity or configure cascade

persist this association in the mapping for example @ManyToOne(..,cascade={"persist"}). If

you cannot find out which entity causes the problem implement

'OC\PlatformBundle\Entity\Image#\_\_toString()' to get a clue.

… c'est que Doctrine ne sait pas quoi faire avec l'entité Imagequi est dans l'entité Advert, car vous ne lui avez pas dit de persister cette entité. Si vous avez bien persisté Advert, vous n'avez rien précisé pour Image et Doctrine est un peu perdu. Pour corriger l'erreur, il faut dire à Doctrine de persister cet objetImage , suivez simplement les indications du message d'erreur :

* Soit vous ajoutez manuellement un$em->persist($advert->getImage())dans le contrôleur, avant le flush();
* Soit, et c'est mieux, vous ajoutez une option à l'annotation@ORM\OneToOne dans l'entité Advert, ce que nous avons fait si vous suivez ce cours depuis le début, comme ceci :

/\*\*

 \* @ORM\OneToOne(targetEntity="OC\PlatformBundle\Entity\Image", cascade={"persist"})

 \*/

private $image;

### Relations multiples, imbriquer un ou plusieurs formualaires :

**Many-To-One ou Many-To-Many**

On prend l’exemple de plusieurs CategoryType dans un AdvertType.

Après avoir expliciter encore une fois la classe :

<?php

// src/OC/PlatformBundle/Form/CategoryType.php

namespace OC\PlatformBundle\Form;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\Extension\Core\Type\TextType;

use Symfony\Component\Form\FormBuilderInterface;

use Symfony\Component\OptionsResolver\OptionsResolver;

class CategoryType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('name', *TextType*::class);

}

public function configureOptions(OptionsResolver $resolver)

{

$resolver->setDefaults(array(

'data\_class' => 'OC\PlatformBundle\Entity\Category'

));

}

}

Maintenant, il faut rajouter le champ categories dans le AdvertType. Il faut pour cela utiliser le type collection et lui passer quelques options, comme ceci :

<?php

// src/OC/PlatformBundle/Form/AdvertType.php

class AdvertType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('date', *DateTimeType*::class)

->add('title', *TextType*::class)

->add('author', *TextType*::class)

->add('content', *TextareaType*::class)

->add('published', *CheckboxType*::class, array('required' => false))

->add('image', *ImageType*::class)

/\*

\* Rappel :

\*\* - 1er argument : nom du champ, ici « categories », car c'est le nom de l'attribut

\*\* - 2e argument : type du champ, ici « CollectionType » qui est une liste de quelque chose

\*\* - 3e argument : tableau d'options du champ

\*/

->add('categories', *CollectionType*::class, array(

'entry\_type' => *CategoryType*::class,

'allow\_add' => true,

'allow\_delete' => true

))

->add('save', *SubmitType*::class);

}

}

On sait maintenant que Categorie est un tableau de Categorie. On a rajouté quelques options après.

On remarque en inspectant l’élément, que rien ne s’affiche, en fait on ne sait pas lequel afficher, il faudrait que chaque élément ait un nom différent. On doit mettre du JS.

On voit ça :

<div class="form-group">

<label class="col-sm-2 control-label required">\_\_name\_\_label\_\_</label>

<div class="col-sm-10">

<div id="advert\_categories\_\_\_name\_\_">

<div class="form-group">

<label class="col-sm-2 control-label required" for="advert\_categories\_\_\_name\_\_\_name">Name</label>

<div class="col-sm-10">

<input type="text" id="advert\_categories\_\_\_name\_\_\_name" name="advert[categories][\_\_name\_\_][name]" required="required" class="form-control" />

</div>

</div>

</div>

</div>

</div>

{# src/OC/PlatformBundle/Resources/views/Advert/form.html.twig #}

{# Le formulaire reste globalement le même,

On ne rajoute que le champ catégorie et le lien pour en ajouter #}

<div class="well">

{# ... #}

{{ form\_row(form.categories) }}

<a href="#" id="add\_category" class="btn btn-default">Ajouter une catégorie</a>

{# ... #}

</div>

{# On charge la bibliothèque jQuery. Ici, je la prends depuis le CDN google

mais si vous l'avez en local, changez simplement l'adresse. #}

<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.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 = $('div#advert\_categories');

// On définit un compteur unique pour nommer les champs qu'on va ajouter dynamiquement

*var* index = $container.find(':input').length;

// On ajoute un nouveau champ à chaque clic sur le lien d'ajout.

$('#add\_category').click(*function*(e) {

addCategory($container);

e.preventDefault(); // évite qu'un # apparaisse dans l'URL

return false;

});

// On ajoute un premier champ automatiquement s'il n'en existe pas déjà un (cas d'une nouvelle annonce par exemple).

if (index == 0) {

addCategory($container);

} else {

// S'il existe déjà des catégories, on ajoute un lien de suppression pour chacune d'entre elles

$container.children('div').each(*function*() {

addDeleteLink($(this));

});

}

// La fonction qui ajoute un formulaire CategoryType

*function* addCategory(*$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* template = $container.attr('data-prototype')

.replace(/\_\_name\_\_label\_\_/g, 'Catégorie n°' + (index+1))

.replace(/\_\_name\_\_/g, index)

;

// On crée un objet jquery qui contient ce template

*var* $prototype = $(template);

// On ajoute au prototype un lien pour pouvoir supprimer la catégorie

addDeleteLink($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* addDeleteLink(*$prototype*) {

// Création du lien

*var* $deleteLink = $('<a href="#" class="btn btn-danger">Supprimer</a>');

// Ajout du lien

$prototype.append($deleteLink);

// Ajout du listener sur le clic du lien pour effectivement supprimer la catégorie

$deleteLink.click(*function*(e) {

$prototype.remove();

e.preventDefault(); // évite qu'un # apparaisse dans l'URL

return false;

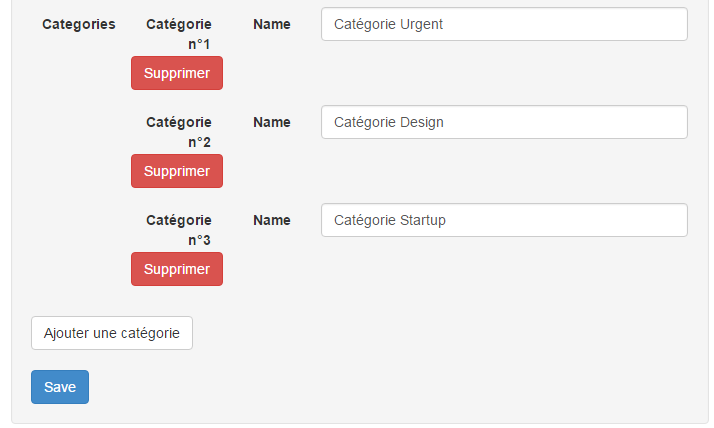
});

}

});

</script>

On obtient maintenant :



### Le type de champ Entity :

Différence entre EntityType et CollectionType : On peut l’utiliser à la place de CollectionType.

<?php

// src/OC/PlatformBundle/Form/AdvertType.php

use Symfony\Bridge\Doctrine\Form\Type\EntityType;

// ...

$builder

->add('categories', *EntityType*::class, array(

'class' => 'OCPlatformBundle:Category',

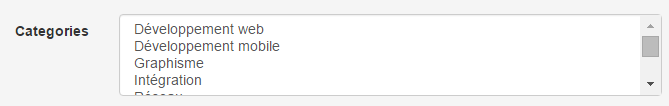
'choice\_label' => 'name',

'multiple' => true,

))

;

On obitent :



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 entitésCategory, on renseigne donc le raccourci Doctrine pour cette entité (ou son namespace complet).
* L'option choice\_label définit comment afficher les entités dans leselectdu 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 optionchoice\_label. Ici j'ai renseignéname, c'est donc via leur nom qu'on liste les catégories dans leselect. Sachez que vous pouvez également renseignerdisplay (ou autre !) et créer le getter associé (à savoirgetDisplay()) dans l'entitéCategory, ce sera donc le retour de cette méthode qui sera affiché dans leselect.
* L'optionmultipledéfinit 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 entitéCategory) et votre entitéAdvert (qui attend une liste d'entitésCategory) ne vont pas s'entendre !

Si la fonctionnalité de ce type (sélectionner une ou plusieurs entités) est unique, le rendu peut avoir quatre formes en fonction des optionsmultipleetexpanded:

Par défaut ils sont à false.

### L’option queryBuilder :

Permet de passer au champ un QueryBuilder. Imaginons qu’on ne veuille que les catégories qui commencent par une certaine lettre : On commence par créer une méthode dans le repository de la catégorie.

<?php

// src/OC/PlatformBundle/Repository/CategoryRepository.php

namespace OC\PlatformBundle\Repository;

use Doctrine\ORM\EntityRepository;

class CategoryRepository extends EntityRepository

{

public function getLikeQueryBuilder($pattern)

{

return $this

->createQueryBuilder('c')

->where('c.name LIKE :pattern')

->setParameter('pattern', $pattern)

;

}

}

Retourne un QueryBuilder et non une Query ou le resultat d’une requête.

Pour que Doctrine utilise votre nouveau repository, n'oubliez pas de rajouter l'option repositoryClass à votre entité Category avec l'annotation suivante :  @ORM\Entity(repositoryClass="OC\PlatformBundle\Repository\CategoryRepository")

Il ne reste maintenant qu'à faire appel à cette méthode depuis l'optionquery\_buildergrâce à une closure dont l'argument est le repository, comme ceci :

<?php

// src/OC/PlatformBundle/Form/AdvertType.php

use OC\PlatformBundle\Repository\CategoryRepository;

class AdvertType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

// Arbitrairement, on récupère toutes les catégories qui commencent par "D"

$pattern = 'D%';

$builder

// ...

->add('categories', *EntityType*::class, array(

'class' => 'OCPlatformBundle:Category',

'choice\_label' => 'name',

'multiple' => true,

'query\_builder' => function(CategoryRepository $repository) use($pattern) {

return $repository->getLikeQueryBuilder($pattern);

}

))

;

}

}

On a passé l’argument pattern, on pourrait en passer plus en mettant une virgule.

## Aller plus loin avec les forumaires :

### Héritage :

Les xxxType sont de simples objets mais Form a un système d’héritage.

**L’héritage avec les forumaires permet de générer des formulaires différents mais avec la même base.**

On va donc faire hériter AdvertEditType de AdvertType. Le processus est le suivant :

1. Copiez-collez le fichier AdvertType.php et renommez la copie en AdvertEditType.php;
2. Modifiez le nom de la classe en AdvertEditType ;
3. Ajouter une méthode getParent  qui retourne la classe du formulaire parent,AdvertType::class  ;
4. Remplacez la définition manuelle de tous les champs (les$builder->add()) par une simple ligne pour supprimer le champ date: $builder->remove('date') ;
5. Enfin, supprimez la méthode configureOptions()qu'il ne sert à rien d'hériter dans notre cas.

Voici ce que cela donne :

<?php

// src/OC/PlatformBundle/Form/AdvertEditType.php

namespace OC\PlatformBundle\Form;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\FormBuilderInterface;

class AdvertEditType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder->remove('date');

}

public function getParent()

{

return *AdvertType*::class;

}

}

La différence entre l’héritage PHP et celle des formulaires est la méthode getParent() qui renvoie le forumaire parent.

Donc ici, on ne pourra pas modifier la date de l’advert.

Si vous voulez utiliser la même vue pour les deux formulaires, vous aurez une erreur comme quoi le champ form.date n'existe pas dans la vue pour le formulaire d'édition. Pour corriger cela, englobez simplement le{{ form\_row(form.date) }} d'un{% if form.date is defined %}.

### Formulaire différent selon des paramètres :

Par exemple, on pourrait empêcher de dépublier une annonce une fois qu'elle est publiée. Le comportement serait le suivant :

* Si l'annonce n'est pas encore publiée, on peut modifier sa valeur de publication lorsqu'on modifie l'annonce.
* Si l'annonce est déjà publiée, on ne peut plus modifier sa valeur de publication lorsqu'on modifie l'annonce.

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():

<?php

// src/OC/PlatformBundle/Form/AdvertType.php

namespace OC\PlatformBundle\Form;

use OC\PlatformBundle\Repository\CategoryRepository;

use Symfony\Bridge\Doctrine\Form\Type\EntityType;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\Extension\Core\Type\CheckboxType;

use Symfony\Component\Form\Extension\Core\Type\DateTimeType;

use Symfony\Component\Form\Extension\Core\Type\SubmitType;

use Symfony\Component\Form\Extension\Core\Type\TextareaType;

use Symfony\Component\Form\Extension\Core\Type\TextType;

use Symfony\Component\Form\FormBuilderInterface;

// N'oubliez pas ces deux use !

use Symfony\Component\Form\FormEvent;

use Symfony\Component\Form\FormEvents;

use Symfony\Component\OptionsResolver\OptionsResolver;

class AdvertType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

// Ajoutez ici tous vos champs sauf le champ published

$builder = ...;

// On ajoute une fonction qui va écouter un évènement

$builder->addEventListener(

*FormEvents*::PRE\_SET\_DATA, // 1er argument : L'évènement qui nous intéresse : ici, PRE\_SET\_DATA

function(FormEvent $event) { // 2e argument : La fonction à exécuter lorsque l'évènement est déclenché

// On récupère notre objet Advert sous-jacent

$advert = $event->getData();

// Cette condition est importante, on en reparle plus loin

if (null === $advert) {

return; // On sort de la fonction sans rien faire lorsque $advert vaut null

}

// Si l'annonce n'est pas publiée, ou si elle n'existe pas encore en base (id est null)

if (!$advert->getPublished() || null === $advert->getId()) {

// Alors on ajoute le champ published

$event->getForm()->add('published', *CheckboxType*::class, array('required' => false));

} else {

// Sinon, on le supprime

$event->getForm()->remove('published');

}

}

);

}

// ...

}

Récupérer l'Advert nous permet d'utiliser les valeurs qu'il contient, chose qu'on ne pouvait 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 duXxxType, pensez à un champ de typeCollectionType, rappelez-vous sa définition :

<?php

$builder->add('categories', *CollectionType*::class, array('entry\_type' => *CategoryType*::class);

Avec ce code, on ne crée qu'*un seul* objetCategoryType, or celui-ci sera utilisé pour ajouter *plusieurs*catégories différentes. Il est donc normal de ne pas avoir accès à l'objet$category lors de la construction du formulaire, autrement dit la construction de l'objetCategoryType. C'est pour cela qu'il faut utiliser l'évènementPRE\_SET\_DATA, qui, lui, est déclenché à chaque fois que le formulaire remplit les valeurs de ses champs par les valeurs d'un nouvel objetCategory.

Je reviens sur la conditionif (null == $advert)dans la fonction. En fait, à la première création du formulaire, celui-ci exécute sa méthodesetData()avecnullen argument. Cette occurrence de l'évènementPRE\_SET\_DATAne nous intéresse pas, d'où la condition pour sortir de la fonction lorsque$event->getData()vautnull. Ensuite, lorsque le formulaire récupère l'objet ($advert dans notre cas) sur lequel se construire, il réexécute sa méthodesetData()avec l'objet en argument. C'est cette occurrence-là qui nous intéresse.

## Type de champ File pour envoyer des fichiers :

### Type de champ File :

**Retourne une instance de la classe UploadedFile. Dans la BDD on ne va stocker que l’adresse du fichier, donc du texte. Il faut utiliser un attribut à part dans l’entité sous-jacente au formulaire, ici Image.**

On rajoute donc File dans l’entité Image.

<?php

// src/OC/PlatformBundle/Entity/Image

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

// N'oubliez pas ce use :

use Symfony\Component\HttpFoundation\File\UploadedFile;

/\*\*

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\ImageRepository")

\*/

class Image

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @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;

public function getFile()

{

return $this->file;

}

public function setFile(UploadedFile $file = null)

{

$this->file = $file;

}

// ...

}

On ne persist pas File donc on ne l’annote pas, il sera utilisé dans le formulaire par contre.

De même dans ImageType : on permet à l’utilisateur de charger un fichier et non donner un URL.

<?php

// src/OC/PlatformBundle/Form/ImageType.php

namespace OC\PlatformBundle\Form;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\Extension\Core\Type\FileType;

use Symfony\Component\Form\FormBuilderInterface;

use Symfony\Component\OptionsResolver\OptionsResolver;

class ImageType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('file', *FileType*::class)

;

}

}

### Manipuler le fichier envoyé :

<?php

// src/OC/PlatformBundle/Entity/Image

class Image

{

public function upload()

{

// Si jamais il n'y a pas de fichier (champ facultatif), on ne fait rien

if (null === $this->file) {

return;

}

// On récupère 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;

}

public function getUploadDir()

{

// On retourne le chemin relatif vers l'image pour un navigateur (relatif au répertoire /web donc)

return 'uploads/img';

}

protected function getUploadRootDir()

{

// On retourne le chemin relatif vers l'image pour notre code PHP

return \_\_DIR\_\_.'/../../../../web/'.$this->getUploadDir();

}

}

D'une part, on a défini le répertoire dans lequel stocker nos images. J'ai mis iciuploads/img, ce répertoire est relatif au répertoireweb, vous pouvez tout à fait le personnaliser. La méthodegetUploadDir()retourne ce chemin relatif, à utiliser dans vos vues car les navigateurs sont relatifs à notre répertoireweb. La méthodegetUploadRootDir(), 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 répertoireweb, il faut remonter pas mal de dossiers, comme vous pouvez le voir. :p

D'autre part, la méthodeupload()s'occupe concrètement de notre fichier. Elle fait l'équivalent dumove\_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 entitéImagedans la base de données, la méthodeupload()s'occupe également de renseigner les deux attributs persistés,$urlet$alt. En effet, l'attribut$file, qui est le seul rempli par le formulaire, n'est pas du tout persisté.

Pour l’executer, côté contrôleur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

// …

public function addAction(Request $request)

{

$advert = new *Advert*();

$form = $this->get('form.factory')->create(*AdvertType*::class, $advert);

if ($request->isMethod('POST') && $form->handleRequest($request)->isValid()) {

// Ajoutez cette ligne :

// c'est elle qui déplace l'image là où on veut les stocker

$advert->getImage()->upload();

// Le reste de la méthode reste inchangé

$em = $this->getDoctrine()->getManager();

$em->persist($advert);

$em->flush();

// ...

}

// ...

}

// …

**On upload avant de persister !**

### Automatisation avec les évènements :

4 actions différentes à exécuter :

* Avant l’enregistrement dans la BDD, on doit renseigner les champs $alt et $url. On utilisera donc **PrePersist** et **PreUpdate**.
* Après l’enregistrement on déplace le fichier, on ne le fait pas avant au cas où l’enregistrement échouerait. On utilise donc **PostPersist** et **PostUpdate**.
* Juste avant la suppression, il faut sauvegarder le nom du fichier dans un attribut non persisté comme $fiename. Comme on n’accède au nom grace à l’id, si on la supprime on n’aura plus accès à l’ID avec un **PostRemove**. On utilisera donc **PreRemove**.
* Juste après la suppression, il faut supprimer le fichier qui était associé à l’entité. On ne le fait pas avant la suppression car si l’entité n’est pas supprimé, on aurait une entité sans fichier. On utilise donc **PostRemove**.

### Implémenter les méthodes des évènements :

* On éclate l’ancien code de la méthode upload() dans les méthodes :
  + preUpload() pour ce qui est de la générations ees attributs $alt et $url
  + upload() pour le déplacement effectif des fichiers.
* On ajoute une méthode preRemoveUpload() qui sauvegarde le fichier qui dépend de l’id de l’entité dans un attribut temporaire.
* On ajoute une méthode remove upload() qui supprime effectivement le fichier grace au nom enregistré.

Ne pas oublier de rajouter un attribut (ici $tempFilename).

**Au final :**

<?php

// src/OC/PlatformBundle/Entity/Image

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

use Symfony\Component\HttpFoundation\File\UploadedFile;

/\*\*

\* @ORM\Table(name="oc\_image")

\* @ORM\Entity

\* @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), on ne fait rien

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), on ne fait rien

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();

}

// …

}

L’upload dans le contrôleur n’est plus utile car on a fait automatiquement avec les évènements.

Pour que l'entitéImageliée à une annonce soit supprimée lorsque vous supprimez l'entitéAdvert, assurez-vous que l'actionremovesoit en cascade. Pour cela, votre annotation sur l'attribut$imagedans votre entitéAdvert devrait ressembler à ceci :@ORM\OneToOne(targetEntity="OC\PlatformBundle\Entity\Image", cascade={"persist", "remove"}).

Vous devez également modifier la vueview.html.twigqui affiche les images. Nous avions utilisé{{ advert.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 :

<img

src="{{ asset(advert.image.uploadDir ~ '/' ~ advert.image.id ~ '.' ~ advert.image.url) }}"

alt="{{ advert.image.alt }}"

/>

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 exemplegetWebPath():

<?php

// src/OC/PlatformBundle/Entity/Image

public function getWebPath()

{

return $this->getUploadDir().'/'.$this->getId().'.'.$this->getUrl();

}

Et du coup, dans la vue, il ne reste plus que :

<img

src="{{ asset(advert.image.webPath) }}"

alt="{{ advert.image.alt }}"

/>

**Le rendu final de l’application sur le cours.**

# Valider ses données

## Pourquoi valider ses données ?

**Never trust user input**.

**VALIDATOR** permet de valider ses données. Un formulaire a besoin de validator mais pas l’inverse, on peut donc l’utiliser ailleurs.

## Définir les règles de validations :

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  Doctrine 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.

On définis quelques règles :

* La date doit être une date valide ;
* Le titre doit faire au moins 10 caractères de long ;
* Le contenu ne doit pas être vide ;
* Le nom de l'auteur doit faire au moins 2 caractères de long ;
* L'image liée doit être valide selon les règles attachées à l'objet Image.

### Annotation :

Pour le mapping, Doctrine était @ORM, ici on utilise les Asserts :

<?php

use Symfony\Component\Validator\Constraints as Assert;

On obtient ainsi :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

use Doctrine\ORM\Mapping as ORM;

use Gedmo\Mapping\Annotation as Gedmo;

// N'oubliez pas de rajouter ce « use », il définit le namespace pour les annotations de validation

use Symfony\Component\Validator\Constraints as Assert;

/\*\*

\* @ORM\Table(name="oc\_advert")

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Repository\AdvertRepository")

\* @ORM\HasLifecycleCallbacks()

\*/

class Advert

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="AUTO")

\*/

private $id;

/\*\*

\* @ORM\Column(name="date", type="datetime")

\* @Assert\DateTime()

\*/

private $date;

/\*\*

\* @ORM\Column(name="title", type="string", length=255)

\* @Assert\Length(min=10)

\*/

private $title;

/\*\*

\* @ORM\Column(name="author", type="string", length=255)

\* @Assert\Length(min=2)

\*/

private $author;

/\*\*

\* @ORM\Column(name="content", type="text")

\* @Assert\NotBlank()

\*/

private $content;

/\*\*

\* @ORM\OneToOne(targetEntity="OC\PlatformBundle\Entity\Image", cascade={"persist", "remove"})

\* @Assert\Valid()

\*/

private $image;

// ...

}

On a utilisé la forme simple, qui est construite comme ça :

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

Ou on peut utiliser la version longue qui permet de personnaliser plusieurs attributs :

@Assert\Contrainte(option1="valeur1", option2="valeur2", …)

Un exemple :

@Assert\Length(min=10, minMessage="Le titre doit faire au moins {{ limit }} caractères.")

Oui, vous pouvez utilisez {{ limit }}, qui est ici la longueur minimum définie dans le l'option "min".

### Les contraintes existantes : attention sensible à la casse !

 :

@Assert\Length(min=10, minMessage="Le titre doit faire au moins {{ limit }} caractères.")

Oui, vous pouvez utilisez {{ limit }}, qui est ici la longueur minimum définie dans le l'option "min".

**Contraintes de bases :**

| **Contrainte** | **Rôle** | **Options** |
| --- | --- | --- |
| [NotBlank](https://symfony.com/doc/current/reference/constraints/NotBlank.html) [Blank](https://symfony.com/doc/current/reference/constraints/Blank.html) | 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](https://symfony.com/doc/current/reference/constraints/True.html) [False](https://symfony.com/doc/current/reference/constraints/False.html) | La contrainte True vérifie que la valeur vaut true, 1 ou "1". La contrainte False vérifie que la valeur vaut false, 0 ou "0". | - |
| [NotNull](https://symfony.com/doc/current/reference/constraints/NotNull.html)  [Null](https://symfony.com/doc/current/reference/constraints/Null.html) | La contrainte NotNull vérifie que la valeur est strictement différente de null. | - |
| [Type](https://symfony.com/doc/current/reference/constraints/Type.html) | 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](https://symfony.com/doc/current/reference/constraints/Type.html#options). |

**Contraintes sur des chaînes de caractères :**

| **Contrainte** | **Rôle** | **Options** |
| --- | --- | --- |
| [Email](https://symfony.com/doc/current/reference/constraints/Email.html) | La contrainte Email vérifie que la valeur est une adresse e-mail valide. | checkMX (défaut : false) : si défini à true, Symfony va vérifier les MX de l'e-mail via la fonction [checkdnsrr](http://www.php.net/manual/en/function.checkdnsrr.php). |
| [Length](https://symfony.com/doc/current/reference/constraints/Length.html) | La contrainte Length vérifie que la valeur donnée fait au moins X ou au plus Y caractères de long. | min : le nombre de caractères minimum à respecter. max : le nombre de catactères maximum à respecter. minMessage : le message d'erreur dans le cas où la contrainte minimum n'est pas respectée. maxMessage : le message d'erreur dans le cas où la contrainte maximum n'est pas respectée. charset (défaut : UTF-8) : le charset à utiliser pour calculer la longueur. |
| [Url](https://symfony.com/doc/current/reference/constraints/Url.html) | La contrainte Url vérifie que la valeur est une adresse URL valide. | protocols (défaut : array('http', 'https')) : définit les protocoles considérés comme valides. Si vous voulez accepter les URL en ftp://, ajoutez-le à cette option. |
| [Regex](https://symfony.com/doc/current/reference/constraints/Regex.html) | La contrainte Regex vérifie la valeur par rapport à une regex. | pattern (option par défaut) : la regex à faire correspondre. match (défaut : true) : définit si la valeur doit (true) ou ne doit pas (false) correspondre à la regex. |
| [Ip](https://symfony.com/doc/current/reference/constraints/Ip.html) | La contrainte Ip vérifie que la valeur est une adresse IP valide. | type (défaut : 4) : version de l'IP à considérer. 4 pour IPv4, 6 pour IPv6, allpour toutes les versions, et [d'autres](https://symfony.com/doc/current/reference/constraints/Ip.html#options). |
| [Language](https://symfony.com/doc/current/reference/constraints/Language.html) | La contrainte Language vérifie que la valeur est un code de langage valide selon [la norme](https://www.loc.gov/standards/iso639-2/php/code_list.php). | - |
| [Locale](https://symfony.com/doc/current/reference/constraints/Locale.html) | La contrainte Locale vérifie que la valeur est une locale valide. Exemple : fr ou fr\_FR. | - |
| [Country](https://symfony.com/doc/current/reference/constraints/Country.html) | La contrainte Country vérifie que la valeur est un code pays en 2 lettres valide. Exemple : fr. | - |

Contraintes sur les nombres :

| **Contrainte** | **Rôle** | **Options** |
| --- | --- | --- |
| [Range](https://symfony.com/doc/current/reference/constraints/Range.html) | La contrainte Range vérifie que la valeur ne dépasse pas X, ou qu'elle dépasse Y. | min : la valeur minimum à respecter. max : la valeur maximum à respecter. minMessage : le message d'erreur dans le cas où la contrainte minimum n'est pas respectée. maxMessage : le message d'erreur dans le cas où la contrainte maximum n'est pas respectée. invalidMessage : message d'erreur lorsque la valeur n'est pas un nombre. |

**Contraintes sur les dates :**

| **Contrainte** | **Rôle** | **Options** |
| --- | --- | --- |
| [Date](https://symfony.com/doc/current/reference/constraints/Date.html) | La contrainte Date vérifie que la valeur est un objet de type Datetime, ou une chaîne de caractères du type YYYY-MM-DD. | - |
| [Time](https://symfony.com/doc/current/reference/constraints/Time.html) | La contrainte Time vérifie que la valeur est un objet de type Datetime, ou une chaîne de caractères du type HH:MM:SS. | - |
| [DateTime](https://symfony.com/doc/current/reference/constraints/DateTime.html) | La contrainte Datetime vérifie que la valeur est un objet de type Datetime, ou une chaîne de caractères du type YYYY-MM-DD HH:MM:SS. | - |

**Contraintes sur les fichiers :**

| **Contrainte** | **Rôle** | **Options** |
| --- | --- | --- |
| [File](https://symfony.com/doc/current/reference/constraints/File.html) | La contrainte File vérifie que la valeur est un fichier valide, c'est-à-dire soit une chaîne de caractères qui pointe vers un fichier existant, soit une instance de la classe [File](https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpFoundation/File/File.php) (ce qui inclut UploadedFile). | maxSize : la taille maximale du fichier. Exemple : 1M ou 1k. mimeTypes : mimeType(s) que le fichier doit avoir. |
| [Image](https://symfony.com/doc/current/reference/constraints/Image.html) | La contrainte Image vérifie que la valeur est valide selon la contrainte précédente File (dont elle hérite les options), sauf que les mimeTypes acceptés sont automatiquement définis comme ceux de fichiers images. Il est également possible de mettre des contraintes sur la hauteur max ou la largeur max de l'image. | maxSize : la taille maximale du fichier. Exemple : 1M ou 1k. minWidth / maxWidth : la largeur minimale et maximale que doit respecter l'image. minHeight / maxHeight : la hauteur minimale et maximale que doit respecter l'image. |

## Déclencher la validation :

### Le service Validator :

On doit déclencher la validation, elle ne se fait pas toute seule. Il s’obtient comme n’importe quel service :

<?php

// Depuis un contrôleur

$validator = $this->get('validator');

**Ensuite on utilise la méthode validate qui renvoie un objet vide si tout va bien, ou remplie avec les erreurs sinon.**

<?php

// Depuis un contrôleur

// …

public function testAction()

{

$advert = new *Advert*;

$advert->setDate(new \Datetime()); // Champ « date » OK

$advert->setTitle('abc'); // Champ « title » incorrect : moins de 10 caractères

//$advert->setContent('blabla'); // Champ « content » incorrect : on ne le définit pas

$advert->setAuthor('A'); // Champ « author » incorrect : moins de 2 caractères

// On récupère le service validator

$validator = $this->get('validator');

// On déclenche la validation sur notre object

$listErrors = $validator->validate($advert);

// Si $listErrors n'est pas vide, on affiche les erreurs

if(count($listErrors) > 0) {

// $listErrors est un objet, sa méthode \_\_toString permet de lister joliement les erreurs

return new *Response*((string) $listErrors);

} else {

return new *Response*("L'annonce est valide !");

}

}

### Validation automatique sur les forumaires :

En réalité, dans :

<?php

if ($form->handleRequest($request)->isValid()) {

// ...

}

HandleRequest va faire appel à validator tout seul, du coup on ne doit pas le faire nous-même, on doit juste définir les Asserts.

## Encore plus de règle de validation :

### Valider depuis un Getter :

On peut mettre une validation sur une fonction comme on accepte sur les Getters.

<?php

class Advert

{

// …

/\*\*

\* @Assert\IsTrue()

\*/

public function isAdvertValid()

{

return false;

}

}

Ici on attend un True sauf qu’on renvoie toujours False. Le message sera en haut du formulaire. Appliqué à l’ensemble du coup. On peut l’appliquer à un champ en particulier :

<?php

class Advert

{

// …

/\*\*

\* @Assert\IsTrue()

\*/

public function isTitle()

{

return false;

}

}

Ici avec isTitle, celui-ci sera automatiquement appliqué sur le champs Title au lieu d’être appliqué à l’ensemble du formulaire. On peut donc faire toute une série de traitement et vérifier le résultat final avec les Asserts.

### Valider intelligemment un attribut objet :

On utilise **Valid()** :

<?php

class A

{

/\*\*

\* @Assert\Length(min=5)

\*/

private $title;

/\*\*

\* @Assert\Valid()

\*/

private $b;

}

class B

{

/\*\*

\* @Assert\Range(max=10)

\*/

private $number;

}

Valider une sous-classe n’est pas le comportement par défaut, on met donc sur le dernière attribut de la première classe : Assert\Valid() qui permet de chercher aussi les erreurs dans la sous-classe.

### Valider depuis un callback() :

L’objectif de callback est de le personnaliser au maximum, on peut vérifier par aute chose que length ou autre.

L’exemple classique : la censure des mots.

Imaginons qu’on ne souhaite pas faire apparaître : démotivation et abandon :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

// Ajoutez ce use pour le contexte

use Symfony\Component\Validator\Context\ExecutionContextInterface;

/\*\*

\* @ORM\Entity

\*/

class Advert

{

// …

/\*\*

\* @Assert\Callback

\*/

public function isContentValid(ExecutionContextInterface $context)

{

$forbiddenWords = array('démotivation', 'abandon');

// On vérifie que le contenu ne contient pas l'un des mots

if (preg\_match('#'.implode('|', $forbiddenWords).'#', $this->getContent())) {

// La règle est violée, on définit l'erreur

$context

->buildViolation('Contenu invalide car il contient un mot interdit.') // message

->atPath('content') // attribut de l'objet qui est violé

->addViolation() // ceci déclenche l'erreur, ne l'oubliez pas

;

}

}

}

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 chacune se trouve grâce à la méthode atPath (en mettant content ou title, etc). Souvent la contrainte sur un getter suffira, mais pensez à ce Callback pour les fois où vous serez limités.

### Valider un champ unique :

On utilise **UniqueEntity** qui n’appartient pas au service Validator mais appartient au Bridge qui fait le lien entre Symfony et Doctrine. On ne peut pas l’utiliser avec Assert comme on fait depuis le début, on est obligé de faire ainsi :

use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

Avec un **USE**. De même, on ne mettra pas Assert\UniqueEntity mais plutôt **@UniqueEntity(field=…)**.

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

// On rajoute ce use pour la contrainte :

use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/\*\*

\* @ORM\Entity

\* @UniqueEntity(fields="title", message="Une annonce existe déjà avec ce titre.")

\*/

class Advert

{

/\*\*

\* @var string

\*

\* Et pour être logique, il faudrait aussi mettre la colonne titre en Unique pour Doctrine :

\* @ORM\Column(name="title", type="string", length=255, unique=true)

\*/

private $title;

// ...

}

**Attention, ici on définit sur la classe et non sur un attribut ou une méthode !**

## Valider selon nos propres contraintes :

Les atoux sont doubles :

* Grace à ça on pourra l’appliquer sur différents objets.
* On place le code dans un service.

**Création d’un antiflood :**

### Gérer la contrainte :

La contrainte est basique, toute la logique sera dans le validateur :

<?php

// src/OC/PlatformBundle/Validator/Antiflood.php

namespace OC\PlatformBundle\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.";

}

@Annotation est nécessaire. Il correspond aux attributs publics, on pourra donc faire :

@Antiflood(message="Mon message personnalisé")

### Le validateur :

Décide par quel validateur il doit se faire valider. Une contrainte Xxx devra être validé par xxxValidator :

<?php

// src/OC/PlatformBundle/Validator/AntifloodValidator.php

namespace OC\PlatformBundle\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 de la contrainte

$this->context->addViolation($constraint->message);

}

}

}

Il y a deux moyens de définir une violation :

* Lorsque vous n'avez que le message de l'erreur à passer, vous pouvez utiliser la méthode addViolation  qu'on a utilisé ici.
* Lorsque vous avez plus, comme dans notre précédent callback où on définissait l'attribut sur laquelle attacher la violation, alors vous pouvez utiliser la méthode buildViolation qu'on a utilisé précédemment.

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 utiliser la deuxième méthode pour définir la violation, comme ceci :

<?php

$this->context

->buildViolation($constraint->message)

->setParameters(array('%string%' => $value))

->addViolation()

;

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 Advert :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use OC\PlatformBundle\Validator\Antiflood;

class Advert

{

/\*\*

\* @Assert\NotBlank()

\* @Antiflood()

\*/

private $content;

// …

}

### 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\_stack et entity\_manager, et en y apposant le tag validator.contraint\_validator. Voici ce que cela donne, dans le fichier services.yml dans votre bundle :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.validator.antiflood: # Le nom du service

class: OC\PlatformBundle\Validator\AntifloodValidator # La classe du service, ici notre validateur déjà créé

arguments: ["@request\_stack", "@doctrine.orm.entity\_manager"] # Les données qu'on injecte au service : la requête et l'EntityManager

tags:

- { name: validator.constraint\_validator, alias: oc\_platform\_antiflood } # C'est avec l'alias qu'on retrouvera le service

### Modification des contraintes :

On veut maintenant que notre contrainte soit validée par le service d’alias oc\_platform.antiflood et non pas par AntofloodValidator. Pour cela on rajoute la méthode validateBy().<?php

// src/OC/PlatformBundle/Validator/Antiflood.php

namespace OC\PlatformBundle\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 'oc\_platform\_antiflood'; // Ici, on fait appel à l'alias du service

}

}

### Modification du validateur :

<?php

// src/OC/PlatformBundle/Validator/AntifloodValidator.php

namespace OC\PlatformBundle\Validator;

use Doctrine\ORM\EntityManagerInterface;

use Symfony\Component\HttpFoundation\RequestStack;

use Symfony\Component\Validator\Constraint;

use Symfony\Component\Validator\ConstraintValidator;

class AntifloodValidator extends ConstraintValidator

{

private $requestStack;

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(RequestStack $requestStack, EntityManagerInterface $em)

{

$this->requestStack = $requestStack;

$this->em = $em;

}

public function validate($value, Constraint $constraint)

{

// Pour récupérer l'objet Request tel qu'on le connait, il faut utiliser getCurrentRequest du service request\_stack

$request = $this->requestStack->getCurrentRequest();

// On récupère l'IP de celui qui poste

$ip = $request->getClientIp();

// On vérifie si cette IP a déjà posté une candidature il y a moins de 15 secondes

$isFlood = $this->em

->getRepository('OCPlatformBundle:Application')

->isFlood($ip, 15) // Bien entendu, il faudrait écrire cette méthode isFlood, c'est pour l'exemple

;

if ($isFlood) {

// C'est cette ligne qui déclenche l'erreur pour le formulaire, avec en argument le message

$this->context->addViolation($constraint->message);

}

}

}

# Securité et gestion des utilisateurs

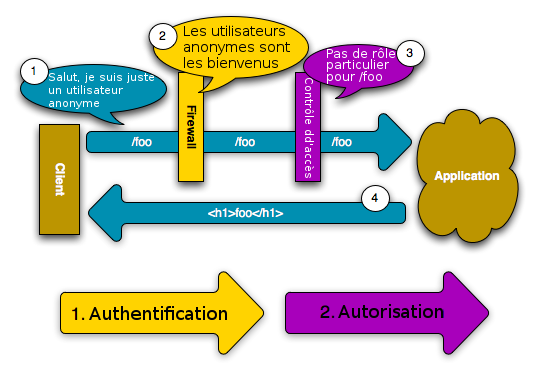
## Authentification et autorisation :

### L’authentification :

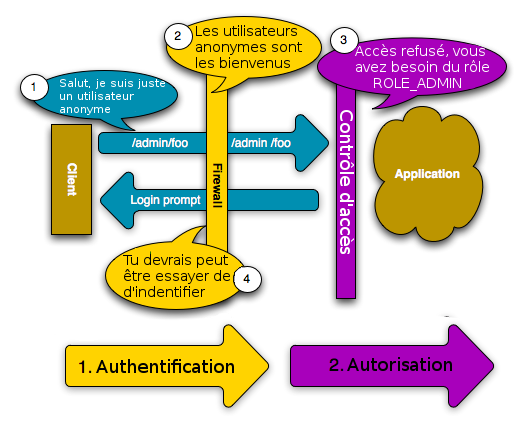
Définit qui on est ent ant que visiteur. Soit on est anonyme, soit on est membre du site. Dans Symfony c’est Firewall qui gère ça. Il peut faire : espace accessible qu’aux membres.

### L’autorisation :

Procéssus qui permet de déterminer si on a le droit d’accéder à une page ou non. Ce qui gère l’autorisation s’appelle l’access control.



Autre exemple : On veut accéder à admin/foo qui demande des droits.



Ici on ne peut pas y accéder, il faut un ROLE\_ADMIN.

## Première approche de la sécurité :

### Fichier de configuration de la sécurité :

Dans app/config on a security.yml.

# app/config/security.yml

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 }

Il s’agit en fait d’un Bundle, SecurityBundle.

**Section encoders**: Objet qui encode les mots de passes. Ici rien n’est encodé : plaintext.

**Setion role\_hierarchy**: Au centre du processus d’autorisation, on assigne un ou plusieurs rôles à chaque utilisateur. Si l’utilisateur dispose des accès, alors il est encodé. Cette section dresse la hérarchie des rôles.

**Section providers** est un fournisseur d’utilisateur. Les firewalls s’adressent aux providers pour récupérer les utilisateurs et les identifier.

**Section firewall :** cherche à vérifier si on est bien celui que l’on prétend être.

**Section access\_control :** différents moyens pour vérifier si on a accès ou non :

* Depuis la configuration en appliquant une règle sur l’URL, en sécurisant une ligne ; ceux qui comment par admin par exemple.
* Ou directement dans le contrôleur : avec ses règles et méthodes.

### Mettre en place un Pare-feu :

Deux choses à mettre en place :

* Construction d’un parfe-feu
* Construction d’un formulaire de connexion.

**Le pare-feu :**

# app/config/security.yml

security:

firewalls:

dev:

pattern: ^/(\_(profiler|wdt)|css|images|js)/

security: false

main:

pattern: ^/

anonymous: true

On rajoute 3 lignes : pattern est le masque d’URL, signifie que toutes les URL commencent par /. Anonymous à true pour dire que l’on accède les gens anonymes.

* 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 nom du fournisseur défini dans la section providers qu'on a vu plus haut.
* 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 définirons 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.

Quand le système de sécurité (ici le parfe-feu, initie le processus d’authentification, il redirige l’utilisateur vers l’espace de connexion (route login). Nom d’utilisateur et MDP du formulaire de connexion seront envoyé par login\_check. On va définir un contrôleur à exécuter pour la route login mais pas pur login\_check :

# app/config/routing.yml

# ...

login:

path: /login

defaults:

\_controller: OCUserBundle:Security:login

login\_check:

path: /login\_check

logout:

path: /logout

En effet, login\_check et logout n’ont pas de contrôleur, ils seront gérés par évènement.

### Créer le Bundle OCUserBundle :

On créer le bundle à l’aide de generate :bundle.

Avant de continuer, je vous propose un petit nettoyage dans ce nouveau OCUserBundle, 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 ;
* La ligne d'import (oc\_user ) du fichier de routes dans le fichier app/config/routing.yml.

### Créer le formulaire de connexion :

On va créer un formulaire de connexion disponible sur la route login soit l’URL /login.

**Le contrôleur :**

<?php

// src/OC/UserBundle/Controller/SecurityController.php;

namespace OC\UserBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

class SecurityController extends Controller

{

public function loginAction(Request $request)

{

// Si le visiteur est déjà identifié, on le redirige vers l'accueil

if ($this->get('security.authorization\_checker')->isGranted('IS\_AUTHENTICATED\_REMEMBERED')) {

return $this->redirectToRoute('oc\_platform\_accueil');

}

// Le service authentication\_utils permet de récupérer le nom d'utilisateur

// et l'erreur dans le cas où le formulaire a déjà été soumis mais était invalide

// (mauvais mot de passe par exemple)

$authenticationUtils = $this->get('security.authentication\_utils');

return $this->render('OCUserBundle:Security:login.html.twig', array(

'last\_username' => $authenticationUtils->getLastUsername(),

'error' => $authenticationUtils->getLastAuthenticationError(),

));

}

}

Le code au milieu n’est là que pour afficher un formulaire et Symfony gère la soumission.

La vue qui est associée :

{# src/OC/UserBundle/Resources/views/Security/login.html.twig #}

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

{% block body %}

{# S'il y a une erreur, on l'affiche dans un joli cadre #}

{% if error %}

<div class="alert alert-danger">{{ 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 %}

On a une erreur dans le DIV quand on rentre un mauvais identifiant.

Ici on a que deux identifiant, ll faut revenir dans le fichier de config pour comprendre. On a défini le fournisseur d’utilisateur de notre pare-feu à **in\_memory**, il lit les utilisateurs directement dans le fichie de configuration, donc deux utilisateurs possibles : **USER** et **ADMIN** avec **USERPASS** et **ADMINPASS**.

### Les erreurs courantes :

* Oublier de créer les routes login, login\_check et logout.
* Les pare-feu ne partagent pas, donc si on est authentifié pour l’un, on ne l’est pas forcément sur un autre.
* Bien mettre login\_check derrière le pare-feu : on doit s’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 l’URL 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, Symfony 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 forcément 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 :

# 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

# ...

### Récupérer l’utilisateur courant :

On utilise security.token\_storage. Ce service est diponible avec getToken() qui permet de récupérer la session de sécurité, à ne pas confondre avec la session classique., disponible avec $request->getSession()).

Le token vaut null si on est hors par-feu et si on est derrière un pare-feu il récupère l’utilisateur courant grace ) $token->getUser().

### Depuis le contrôleur ou un service :

<?php

// On récupère le service

$security = $container->get('security.token\_storage');

// 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();

Pas mal de vérification à faire, mais le contrôleur dispose d’un raccourci :$this->getUser(), elle retourne NULL si pas derière un pare-feu, une instance de User sinon.

<?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 :

On utilise la variable globale {{ app }}

Bonjour {{ app.user.username }} - {{ app.user.email }}

## Gestion des autorisations avec les rôles :

Maintenant on a une authentification fonctionnelle avec : un pare-feu, une méthode d’authentification par formulaire et deux utilisateurs.

### Définition des rôles :

Pour /admin, ça revient à faire, autorisé que pour ROLE\_ADMIN.

Tout d'abord, essayons d'imaginer les rôles dont on aura besoin dans notre application de plateforme d'annonce. Je pense à :

* ROLE\_AUTEUR : pour ceux qui ont le droit d'écrire des annonces ;
* ROLE\_MODERATEUR : pour ceux qui peuvent modérer les annonces ;
* 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 une annonce ». 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.

C’est ce qu’on va écrire dans security.yml

# app/config/security.yml

security:

role\_hierarchy:

# Un admin hérite des droits d'auteur et de modérateur

ROLE\_ADMIN: [ROLE\_AUTEUR, ROLE\_MODERATEUR]

# On garde ce rôle superadmin, il nous resservira par la suite

ROLE\_SUPER\_ADMIN: [ROLE\_ADMIN, ROLE\_ALLOWED\_TO\_SWITCH]

 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 une annonce. Les auteurs et les admins peuvent donc le faire.
* Si l'utilisateur a le rôle ROLE\_ADMIN, alors il peut supprimer une annonce. Seuls les admins peuvent donc le faire.

Rappel, on ne définit ici que la hiérarchie des rôles.

### Le rôle de l’utilisateur :

Il existe quatre méthodes pour faire ce test : les annotations, le service security.authorization\_checker, Twig, et les contrôles d'accès. Ce sont quatre façons de faire exactement la même chose.

#### Utiliser directement le service security.authorization\_checker :

C’est celui par lequel passe les trois autres méthodes. On appelle le service, et on appelle la méthode isGranted().

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\Security\Core\Exception\AccessDeniedException;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// On vérifie que l'utilisateur dispose bien du rôle ROLE\_AUTEUR

if (!$this->get('security.authorization\_checker')->isGranted('ROLE\_AUTEUR')) {

// Sinon on déclenche une exception « Accès interdit »

throw new *AccessDeniedException*('Accès limité aux auteurs.');

}

// Ici l'utilisateur a les droits suffisant,

// on peut ajouter une annonce

}

}

#### Utiliser les annotations :

L’utilisation de @Security provient du Bundle **SensioFrameworkExtraBundle** :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

// N'oubliez pas ce use pour l'annotation

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

class AdvertController extends Controller

{

/\*\*

\* @Security("has\_role('ROLE\_AUTEUR')")

\*/

public function addAction(Request $request)

{

// 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

}

}

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.

#### Depuis une vue Twig :

Pratique pour changer d’affichage en fonction du rôle :

{# On n'affiche le lien « Ajouter une annonce » qu'aux auteurs

(et admins, qui héritent du rôle auteur) #}

{% if is\_granted('ROLE\_AUTEUR') %}

<li><a href="{{ path('oc\_platform\_add') }}">Ajouter une annonce</a></li>

{% endif %}

#### Utiliser les contrôles d’accès :

Permet de sécuriser des URL, elle se configure dans le fichier de configuration de la sécurité. Voici comment sécuriser tout un panel /admin :

# app/config/security.yml

security:

access\_control:

- { path: ^/admin, roles: 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 :

# app/config/security.yml

security:

access\_control:

- { path: ^/admin, ip: 127.0.0.1, requires\_channel: https }

**Annotation sécurise une méthode.**

**Twig sécurise l’affichage.**

**Contrôles d’accès sécurisent l’URL**

## Utiliser des utilisateurs de la BDD :

**Dans Symfony, un utilisateur est un objet qui implémente UserInterface.**

**Il existe aussi une classe User qui implémente cette interface.**

### Créer la classe utilisateur :

On va donc générer une entité User au sein de OCUserBundle. On le crée avec php bin/console doctrine :generate :entity.

* 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 :

<?php

namespace OC\UserBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Table(name="oc\_user")

\* @ORM\Entity(repositoryClass="OC\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 = array();

// Les getters et setters

public function eraseCredentials()

{

}

}

Et pour que Symfony l'accepte comme classe utilisateur de la couche sécurité, il faut qu'on implémente l'interface UserInterface :

<?php

// src/OC/UserBundle/Entity/User.php

use Symfony\Component\Security\Core\User\UserInterface;

class User implements UserInterface

{

// …

}

### Des utilisateurs de test :

On utilise les fixtures :

<?php

// src/OC/UserBundle/DataFixtures/ORM/LoadUser.php

namespace OC\UserBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;

use Doctrine\Common\Persistence\ObjectManager;

use OC\UserBundle\Entity\User;

class LoadUser implements FixtureInterface

{

public function load(ObjectManager $manager)

{

// Les noms d'utilisateurs à créer

$listNames = array('Alexandre', 'Marine', 'Anna');

foreach ($listNames as $name) {

// On crée l'utilisateur

$user = new *User*;

// Le nom d'utilisateur et le mot de passe sont identiques pour l'instant

$user->setUsername($name);

$user->setPassword($name);

// On ne se sert pas du sel pour l'instant

$user->setSalt('');

// On définit uniquement le role ROLE\_USER qui est le role de base

$user->setRoles(array('ROLE\_USER'));

// On le persiste

$manager->persist($user);

}

// On déclenche l'enregistrement

$manager->flush();

}

}

Ensuite on exécute la commande :

php bin/console doctrine:fixtures:load

### Définir l’encoder pour une nouvelle classe utilisateur :

L’encoder pour nos anciens utilisateurs était User, or maintenant il s’agit de OS\UserBundle\Entity\User. On change donc l’encoder pour qu’il corresponde.

# app/config/security.yml

security:

encoders:

Symfony\Component\Security\Core\User\User: plaintext

OC\UserBundle\Entity\User: plaintext

### Définir le fournisseur utilisateur :

Il faut définir un fournisseur d’utilisateur (provider) pour que le pare-feu puisse identifier et récupérer les utilisateurs.

C’est une classe qui implémente **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.

Symfony dispose déjà de trois types de fournisseurs, qui implémentent tous l'interface précédente évidemment, les voici :

* [memory](https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Security/Core/User/InMemoryUserProvider.php) utilise les utilisateurs définis dans la configuration, c'est celui qu'on a utilisé jusqu'à maintenant ;
* [entity](https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Doctrine/Security/User/EntityUserProvider.php) 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 :

 On va créer notre fournisseur entity pour notre entité User :

# app/config/security.yml

security:

providers:

# … vous pouvez supprimer le fournisseur « in\_memory »

# Et voici notre nouveau fournisseur :

main:

entity:

class: OC\UserBundle\Entity\User

property: username

Il y a deux paramètres à préciser pour le fournisseur :

* La classe de l'entité à utiliser évidemment, il s'agit pour le fournisseur de savoir quel repository Doctrine utiliser pour ensuite charger nos entités. Vous pouvez également utiliser le nom logique de l'entité, ici OCUserBundle:User ;
* L'attribut de la classe qui sert d'identifiant, on utilise username, donc on le lui dit.

Main est le nom du provider et entity est le type de provider !

### Dire au pare-feu d’utiliser le nouveau provider :

On préciser qu’on veut utiliser main et non in\_memory défini de base :

# app/config/security.yml

security:

firewalls:

main:

pattern: ^/

anonymous: true

provider: main # On change cette valeur

# … reste de la configuration du pare-feu

### Manipuler vos utilisateurs :

On doit faire un cache :clear avant de manipuler. On peut utiliser FOSUserBundle pour faire des actions de bases avec nos utilisateurs.

## Utiliser FOSUserBundle :

Agit sur l’authentification des membres du site. Il est disponible [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 :

// composer.json

{

// …

"require": {

// …

"friendsofsymfony/user-bundle": "dev-master"

}

// …

}

Ensuite on dit à composer.phar de faire un update du Bundle :

php composer.phar update friendsofsymfony/user-bundle

Et enfin rajouter :

<?php

// app/AppKernel.php

public function registerBundles()

{

$bundles = array(

// …

new *FOS*\UserBundle\FOSUserBundle(),

);

}

Dans app/appKernel.php.

Maintenant le site n’est plus fonctionnel … Un peu de configuration est nécessaire :

### Hériter FOSUserBundle depuis notre OCUserBundle :

Commençons par modifier OCUserBundle :

<?php

// src/OC/UserBundle/OCUserBundle.php

namespace OC\UserBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class OCUserBundle extends Bundle

{

public function getParent()

{

return 'FOSUserBundle';

}

}

Et c’est tout. Lorsque qu'un bundle A (notre OCUserBundle) hérite d'un bundle B (FOSUserBundle), cela signifie entre autre que :

* si une vue du bundle A a le même nom qu'une vue du bundle B, c'est la vue du bundle A qui sera utilisée lorsque vous faites "BundleB::myView.html.twig", alors que vous mentionnez bien "BundleB" dans le nom de la vue ;
* si un contrôleur du bundle A a le même nom qu'un contrôleur du bundle B, c'est le contrôleur du bundle A qui sera utilisé lorsque vous faites "BundleB:myController:myAction", alors que vous mentionnez bien "BundleB" dans le nom du contrôleur.

### Modifier notre entité User :

FOSUser contient une entité User plus complète que la nôtre. On va hériter l’entité User de FOSUserBundle plutôt que de recoder. On va donc hériter de User :

<?php

// src/OC/UserBundle/Entity/User.php

namespace OC\UserBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

use FOS\UserBundle\Model\User as BaseUser;

/\*\*

\* @ORM\Table(name="oc\_user")

\* @ORM\Entity(repositoryClass="OC\UserBundle\Repository\UserRepository")

\*/

class User extends BaseUser

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="AUTO")

\*/

protected $id;

}

Plus besoin d'implémenter UserInterface, car on hérite de l'entité User du bundle FOSUB, qui, elle, implémente cette interface. Pas besoin non plus d'écrire tous les setters et getters, ils sont tous hérités, même le getter getId !

En réaliser User est une **MAPPED SUPERCLASS** elle est abstraite et doit être hérité pour pouvoir être instancié. Elle contient :

* 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.

On peut maintenant rajouter d’autres attributs à notre User.

### Configurer le Bundle :

Dans app/config/config.yml :

# 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: OC\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.

php bin/console doctrine:schema:update --force

### Configuration de la sécurité pour utiliser le Bundle :

On retravaille dans security.yml :

**L’encodeur :**

On ne stock plus les MDP en clair !

# app/config/security.yml

security:

encoders:

OC\UserBundle\Entity\User: sha512

**Le fournisseur :**

Ce Bundle utilise son propre fourisseur :

# 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 :**

Il est déjà à jour car on n’a pas modifié le nom du fournisseur. On rajoute quand même l’option « se souvenir de moi » :

# 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: /platform

remember\_me:

secret: %secret% # %secret% est un paramètre de parameter

%secret% est un paramètre de parameter.

Pour tester, on ne va plus utiliser les fixtures mais plutôt un outil de FOSUserBundle :

php bin/console fos:user:create

### Configuration de FOSUserBundle :

**Les routes :** FOSUser définit déjà des vues, il faut les rajouter aux routes :

# 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

Les routes sont ici en XML et non en YML comme on fait d’habitude.

Voici quand même un extrait de la commande php bin/console debug:router pour les routes qui concernent ce bundle :

fos\_user\_security\_login           ANY      ANY  /login

fos\_user\_security\_check           ANY      ANY  /login\_check

fos\_user\_security\_logout          ANY      ANY  /logout

fos\_user\_profile\_show             GET      ANY  /profile/

fos\_user\_profile\_edit             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 :

# 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: /platform

remember\_me:

secret: %secret%

### Personnalisation esthétique du Bundle :

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/FOSUserBundle/blob/master/Resources/views/layout.html.twig)

On va donc tout simplement le remplacer par une vue Twig qui va étendre notre layout à nous. 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 :

{# src/OC/UserBundle/Resources/views/layout.html.twig #}

{# On étend notre layout #}

{% extends "OCCoreBundle::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, messages in app.session.flashbag.all() %}

{% for message in messages %}

<div class="alert alert-{{ key }}">

{{ message|trans({}, 'FOSUserBundle') }}

</div>

{% endfor %}

{% 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 %}

### Traduire les messages :

C’est un Bundle internationnal, on peut traduire en modifiant le fichier config.yml :

# app/config/config.yml

framework:

translator: { fallbacks: ["%locale%"] }

Il suffit de décommenter ces lignes. Où %local% peut être défini à fr plus haut.

### Afficher une barre utilisateur :

{% 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 %}

A placer dans notre layout.

### Manipuler les utilisateurs avec FOSUserBundle :

Même s’ils sont générés par notre Bundle, nos utilisateurs ne restent que des entités Doctrine. On peut donc créer un repository et on a aussi accès à UserManager qui est une sorte de super repository :

<?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('OCUserBundle:User').

[la documentation officielle sur la sécurité](https://symfony.com/doc/current/book/security.html)

[la documentation de FOSUserBundle](https://symfony.com/doc/master/bundles/FOSUserBundle/index.html)

[utilisation des groupes](https://symfony.com/doc/master/bundles/FOSUserBundle/groups.html).

* 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.
* Le code du cours tel qu'il doit être à ce stade est disponible sur la branche [iteration-16](https://github.com/winzou/mooc-symfony/tree/iteration-16) du dépot Github.

# Les services utilisation poussée

## Les tags sur les serices :

**Concrètement, un tag est une information que l'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.**

### Comprendre les tags à travers Twig :

L'idée est que Twig définit un tag, dans notre cas twig.extension, et une interface, dans notre cas \Twig\_ExtensionInterface. Ensuite, Twig récupère tous les services qui ont ce tag, et sait les utiliser car ils implémentent tous la même interface (ils ont donc des méthodes connues).

### Appliquer un tag à un service :

On pourrait appliquer à nos vues notre service antispam. L'objectif est donc d'avoir dans nos vues une fonction {{ checkIfSpam("mon message") }}

On va donc créer un miniService pour l’utiliser au lieu d’utiliser directement l’autre service, car celui-ci a déjà une mission. Il faut respecter 1 service = une mission, celui-ci doit détecter les spam avec une extenssion twig :

<?php

// src/OC/PlatformBundle/Twig/AntispamExtension.php

namespace OC\PlatformBundle\Twig;

use OC\PlatformBundle\Antispam\OCAntispam;

class AntispamExtension

{

/\*\*

\* @var OCAntispam

\*/

private $ocAntispam;

public function \_\_construct(OCAntispam $ocAntispam)

{

$this->ocAntispam = $ocAntispam;

}

public function checkIfArgumentIsSpam($text)

{

return $this->ocAntispam->isSpam($text);

}

}

Ainsi que sa configuration de base :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.twig.antispam\_extension:

class: OC\PlatformBundle\Twig\AntispamExtension

arguments:

- "@oc\_platform.antispam"

Voilà, nous pouvons maintenant travailler sur ce service.

Commençons donc par ajouter le tag dans sa configuration.

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.twig.antispam\_extension:

class: OC\PlatformBundle\Twig\AntispamExtension

arguments:

- "@oc\_platform.antispam"

tags:

- { name: twig.extension }

On peut ajouter plusieurs tags, d’où le tiret.

### Une classe qui implémente une interface :

Celui qui va récupérer les classes tagés attend un certain comportement de ceux-ci. Il attend que nos services implémentent twigExtensionInterface. Mais Twig propose une classe abstraite à hériter à nos services : Twig\_Extension qui implémente elle-même l’interface.

On modifie donc notre nouveau service comme ainsi :

<?php

// src/OC/PlatformBundle/Twig/AntispamExtension.php

namespace OC\PlatformBundle\Twig;

use OC\PlatformBundle\Antispam\OCAntispam;

class AntispamExtension extends \Twig\_Extension

{

// …

}

Pour qu’il hérite de l’interface.

### Hérite le code qui sera exécuté :

Cette section est propre à chaque tag où celui qui va récupérer les services utilise certaines méthodes. Twig utilise :

* 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 :

<?php

// src/OC/PlatformBundle/Twig/AntispamExtension.php

namespace OC\PlatformBundle\Twig;

use OC\PlatformBundle\Antispam\OCAntispam;

class AntispamExtension extends \Twig\_Extension

{

// ...

// Twig va exécuter cette méthode pour savoir quelle(s) fonction(s) ajoute notre service

public function getFunctions()

{

return array(

new \Twig\_SimpleFunction('checkIfSpam', array($this, 'checkIfArgumentIsSpam')),

);

}

// La méthode getName() identifie votre extension Twig, elle est obligatoire

public function getName()

{

return 'OCAntispam';

}

}

Dans cette méthode getFunctions() , il faut retourner un tableau d'objet \Twig\_SimpleFunction, dont :

* Le premier argument, ici checkIfSpam, est le nom de la fonction qui sera disponible dans nos vues Twig ;
* Le deuxième argument, ici array($this, 'checkIfArgumentIsSpam')  est un callable PHP. Dans notre cas, on appelle la méthode de l'extension Twig même, mais en réalité on pourrait également définir le callable à array($this->ocAntispam, 'isSpam')  pour qu'il appelle directement la méthode de notre service OCAntispam . Les deux méthodes sont possibles ici.

Au final, {{ checkIfSpam(var) }}  côté Twig exécute $this->isSpam($var)  côté OCAntispam .

### Méthodologie :

Tous les tags ne nécessite pas forcément d’implémenter une interface. Pour connaître tous les services implémentant un certain tag, vous pouvez exécuter la commande suivante :

C:\wamp\www\Symfony> php bin/console debug:container --tag=twig.extension

Symfony Container Public Services Tagged with "twig.extension" Tag

==================================================================

------------------------------------- ------------------------------------------

Service ID Class name

------------------------------------- ------------------------------------------

oc\_platform.twig.antispam\_extension OC\PlatformBundle\Twig\AntispamExtension

------------------------------------- ------------------------------------------

‌

### Les principaux tags :

Ils sont tous expliqués [ici](https://symfony.com/fr/doc/current/reference/dic_tags.html).

### Les types de champs du 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](http://ckeditor.com/) 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 :

<?php

// src/OC/PlatformBundle/Form/CkeditorType.php

namespace OC\PlatformBundle\Form;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\Extension\Core\Type\TextareaType;

use Symfony\Component\OptionsResolver\OptionsResolver;

class CkeditorType extends AbstractType

{

public function configureOptions(OptionsResolver $resolver)

{

$resolver->setDefaults(array(

'attr' => array('class' => 'ckeditor') // On ajoute la classe CSS

));

}

public function getParent() // On utilise l'héritage de formulaire

{

return *TextareaType*::class;

}

}

Puis, déclarons cette classe en tant que service, en lui ajoutant le tag form.type :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.form.ckeditor:

class: OC\PlatformBundle\Form\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 AdvertType :

<?php

// src/OC/PlatformBundle/Form/ArticleType.php

namespace OC\PlatformBundle\Form;

class AdvertType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

// …

->add('content', *CkeditorType*::class)

;

}

// …

}

## Dépendances optionnelles : les appels de méthode :

### 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, et choisissons de définir l'argument $locale  comme optionnel. L'idée est de supprimer ce dernier des arguments du constructeur, et d'ajouter le setter correspondant :

<?php

// src/OC/PlatformBundle/Antispam/OCAntispam.php

namespace OC\PlatformBundle\Antispam;

class OCAntispam

{

private $mailer;

private $locale;

private $minLength;

public function \_\_construct(\Swift\_Mailer $mailer, $minLength)

{

$this->mailer = $mailer;

$this->minLength = (int) $minLength;

}

public function setLocale($locale)

{

$this->locale = $locale;

}

// …

}

### Les calls :

Sont un moyen d’exécuter une méthode juste après sa création :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.antispam:

class: OC\PlatformBundle\Antispam\OCAntispam

arguments:

- "@mailer"

- 50

calls:

- [ setLocale, [%locale%] ]

Revient à faire :

<?php

$antispam = new \OC\PlatformBundle\Antispam\OCAntispam($mailer, 50);

$antispam->setLocale($locale);

## Les services courants de Symfony :

| **Identifiant** | **Description** |
| --- | --- |
| doctrine.orm.entity\_manager | Ce service est l'instance de l'EntityManager de Doctrine ORM. On l'a rapidement évoqué dans la partie sur Doctrine, l'EntityManager est bien enregistré en tant que service dans Symfony. Ainsi, lorsque dans un contrôleur vous faites $this->getDoctrine()->getManager(), vous récupérez en réalité le service doctrine.orm.entity\_manager. 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 ! |
| event\_dispatcher | 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. ;) |
| kernel | 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 [Kernel.php](https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpKernel/Kernel.php) pour connaître toutes les possibilités. Nous nous en servirons très peu en réalité. |
| logger | Ce service gère les logs de votre application. Grâce à lui, vous pouvez utiliser des fichiers de logs très simplement. Symfony utilise la classe Monolog par défaut pour gérer ses logs. [La documentation à ce sujet](https://symfony.com/doc/current/cookbook/logging/monolog.html) 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. |
| mailer | Ce service vous renvoie par défaut une instance de Swift\_Mailer, une classe permettant d'envoyer des e-mails facilement. Encore une fois, la [documentation de SwiftMailer](http://swiftmailer.org/docs/introduction.html) et la [documentation de son intégration dans Symfony](http://symfony.comdoc/current/cookbook/email/index.html) vous seront d'une grande aide si vous souhaitez envoyer des e-mails. |
| request\_stack | Ce service est très important : il vous donne un objet qui vous permet de récupérer la requête [Request](https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpFoundation/Request.php) courante via sa méthode getCurrentRequest. Je vous réfère au [chapitre sur les contrôleurs, section Request](https://openclassrooms.com/courses/developpez-votre-site-web-avec-le-framework-symfony2/les-controleurs-avec-symfony2#/id/r-2079456) pour plus d'informations sur comment récupérer la session, l'IP du visiteur, la méthode de la requête, etc.  Ce service utiliser les calls pour définir la requête justement, car la requête peut changer au cours de son exécution (lors d'une sous-requête). |
| router | Ce service vous donne accès au routeur ([Symfony\Component\Routing\Router](https://github.com/fabpot/symfony/blob/master/src/Symfony/Component/Routing/Router.php)). On l'a déjà abordé dans [le chapitre sur les routes](https://openclassrooms.com/courses/developpez-votre-site-web-avec-le-framework-symfony2/le-routeur-de-symfony2), c'est le service qui permet de générer des routes. |
| security.token\_storage | Ce service permet de gérer l'authentification sur votre site internet. On l'utilise notamment pour récupérer l'utilisateur courant. Le raccourci du contrôleur $this->getUser() exécute en réalité $this->container->get('security.token\_storage')->getToken()->getUser() ! |
| service\_container | 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. |
| twig | Ce service représente une instance de [Twig\_Environment](https://github.com/twigphp/Twig/blob/master/lib/Twig/Environment.php). Il permet d'afficher ou de retourner une vue. Vous pouvez en savoir plus en lisant la [documentation de Twig](http://twig.sensiolabs.org/documentation). 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 Symfony. 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 templatinget 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 $this->render() exécute en réalité $this->container->get('templating')->renderResponse(). |

#### 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 principaux noms de services sont à connaître par cœur afin d'injecter ceux nécessaires dans les services que vous créerez !
* Le code du cours tel qu'il doit être à ce stade est disponible sur la branche [iteration-17](https://github.com/winzou/mooc-symfony/tree/iteration-17) du dépot Github.

# Le gestionnaire d’évènement de Symfony

## Des évènements, pour quoi faire ?

Il en existe plusieurs, comme kernel.request par exemple, qui est déclenché avant que le contrôleur ne soit exécuté. Tous les évènements sont déclenchés à un moment clé de l’exécution de la page.

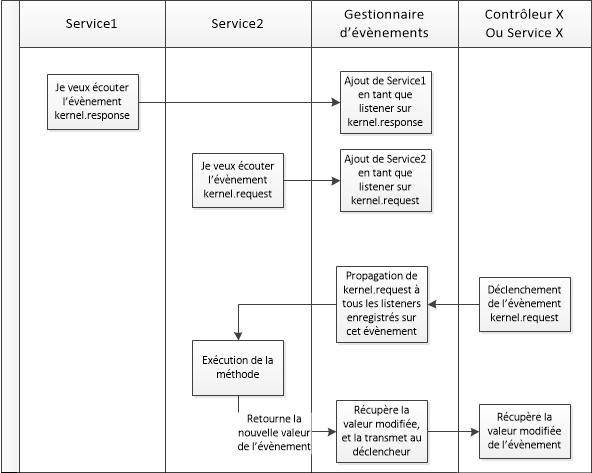
### Qu’est ce qu’un gestionnaire d’évènement ?

**Un découplage**, c’est la principale raison de l’utilisation de gestionnaire :  **Celui qui déclenche l’évènement ne dépend pas du tout de celui qui le réalise.**

On parle de déclencheur d’évènement quand on signal au gestionnaire d’évènement : »tel évènement vient d’arriver, préviens tout le monde ! »

On parle d’écouteur d’évènement lorsqu’on signal au gestionnaire d’évènement « je veux que tu me previennes dès que quelque chose arrive ».

**Un service qui écoute un évènement est appelé un LISTENER.**



On « s’inscrit » sur un évènement et dès qu’il surgit, on applique notre service.

## Ecouter des évènements :

### Créer un service et son listener :

Nous voulons effectuer une certaine action lors d'un certain évènement. Dans ce que nous voulons, il y a deux choses à distinguer :

* D'une part, l'action à réaliser effectivement. Dans notre cas, il s'agit d'ajouter une mention beta à une réponse contenant du HTML. C'est une action certes simple, mais une action quand même, qui mérite son propre objet : un simple service. Appelons-le BetaHTMLAdder car il ajoute la mention beta à du HTML.
* D'autre part, le fait d'exécuter l'action précédente à un certain moment, avec certains paramètres. C'est une autre action, qui mérite donc un autre objet : le listener. Appelons-le BetaListener.

On va d’abords créer la première classe :

<?php

// src/OC/PlatformBundle/Beta/BetaHTMLAdder.php

namespace OC\PlatformBundle\Beta;

use Symfony\Component\HttpFoundation\Response;

class BetaHTMLAdder

{

// Méthode pour ajouter le « bêta » à une réponse

public function addBeta(Response $response, $remainingDays)

{

$content = $response->getContent();

// Code à rajouter

// (Je mets ici du CSS en ligne, mais il faudrait utiliser un fichier CSS bien sûr !)

$html = '<div style="position: absolute; top: 0; background: orange; width: 100%; text-align: center; padding: 0.5em;">Beta J-'.(int) $remainingDays.' !</div>';

// Insertion du code dans la page, au début du <body>

$content = str\_replace(

'<body>',

'<body> '.$html,

$content

);

// Modification du contenu dans la réponse

$response->setContent($content);

return $response;

}

}

Ainsi que la configuration du service :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.beta.html\_adder:

class: OC\PlatformBundle\Beta\BetaHTMLAdder

Le listener :

<?php

// src/OC/PlatformBundle/Beta/BetaListener.php

namespace OC\PlatformBundle\Beta;

use Symfony\Component\HttpFoundation\Response;

class BetaListener

{

// Notre processeur

protected $betaHTML;

// 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 $endDate;

public function \_\_construct(BetaHTMLAdder $betaHTML, $endDate)

{

$this->betaHTML = $betaHTML;

$this->endDate = new \Datetime($endDate);

}

public function processBeta()

{

$remainingDays = $this->endDate->diff(new \Datetime())->days;

if ($remainingDays <= 0) {

// Si la date est dépassée, on ne fait rien

return;

}

// Ici on appelera la méthode

// $this->betaHTML->addBeta()

}

}

**Le rôle du listener est de décider s’il faut appeler ou non un autre objet qui remplira une certaine fonction.**

Il ne fait pas que de décider quand appeler, mais aussi de savoir si on doit l’appeler à chaque fois que l’évènement est déclenché ou s’il y a d’autres paramètres à prendre en compte.

### Ecouter un évènement :

Pour écouter un évènement il faut le présenter au gestionnaire d’évènement.

Deux façons de faire :

* Manipuler directement le gestionnaire d’évènement
* Passer par des services (le plus utilisé)

**Méthode une :**

Consiste à faire passer notre BetaListener au gestionnaire d’évènement. Il existe sous Symfony en tant que EventDispatcher.

<?php

// Depuis un contrôleur

use OC\PlatformBundle\Beta\BetaListener;

// …

// On instancie notre listener

$betaListener = new *BetaListener*('2016-06-01');

// On récupère le gestionnaire d'évènements, qui heureusement est un service !

$dispatcher = $this->get('event\_dispatcher');

// On dit au gestionnaire d'exécuter la méthode onKernelResponse de notre listener

// lorsque l'évènement kernel.response est déclenché

$dispatcher->addListener(

'kernel.response',

array($betaListener, 'processBeta')

);

**Méthode 2 :**

Mettre notre Listener dans un Service.

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.beta.listener:

class: OC\PlatformBundle\Beta\BetaListener

arguments:

- "@oc\_platform.beta.html\_adder"

- "2017-06-01"

Maintenant le Listener est accessible via le conteneur de service. ON définit le tag kernel.event\_listener sur ce service.

Une fois le gestionnaire d'évènements 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 ! 🡺

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.beta.listener:

class: OC\PlatformBundle\Beta\BetaListener

arguments:

- "@oc\_platform.beta.html\_adder"

- "2017-06-01"

tags:

- { name: kernel.event\_listener, event: kernel.response, method: processBeta }

* 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.

Si on veut écouter plusieurs évènements :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.beta.listener:

class: OC\PlatformBundle\Beta\BetaListener

arguments:

- "@oc\_platform.beta.html\_adder"

- "2017-06-01"

tags:

- { name: kernel.event\_listener, event: kernel.response, method: processBeta }

- { name: kernel.event\_listener, event: kernel.controller, method: ignoreBeta }

### Création de la méthode à exécuter avec le Listener :

D’un côté on a notre service BetaHTMLAdder qui réalise la fonction « ajouter la mention Beta ». De l’autre on a la solution technique pour exécuter la fonction quand on le veut, c’est notre Listener Betalistener.

Le listener permet également de passer les bons arguments à notre service BetaHTMLAdder. En effet, lorsque le gestionnaire d'évènements exécute ses listeners, il ne se préoccupe pas de leurs arguments ! Le premier argument qu'il leur donne est 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 Symfony ainsi que les types d'argument que le gestionnaire d'évènements transmet. Vous n'avez pas à jouer aux devinettes !

Dans notre cas, l'évènement FilterResponseEvent dispose des méthodes suivantes :

<?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();

}

Ce sont surtout get et setReponse qui nous sont utile. On a maintenant les infos nécessaires, on peut construire la méthode processBeta :

<?php

// src/OC/PlatformBundle/Beta/BetaListener.php

namespace OC\PlatformBundle\Beta;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class BetaListener

{

// L'argument de la méthode est un FilterResponseEvent

public function processBeta(FilterResponseEvent $event)

{

// On teste si la requête est bien la requête principale (et non une sous-requête)

if (!$event->isMasterRequest()) {

return;

}

// On récupère la réponse que le gestionnaire 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);

}

}

Adapter cette base à notre exemple, on rajoute l’appel à notre service BetaHTMLAdder :

<?php

// src/OC/PlatformBundle/Beta/BetaListener.php

namespace OC\PlatformBundle\Beta;

use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class BetaListener

{

public function processBeta(FilterResponseEvent $event)

{

if (!$event->isMasterRequest()) {

return;

}

$remainingDays = $this->endDate->diff(new \Datetime())->days;

// Si la date est dépassée, on ne fait rien

if ($remainingDays <= 0) {

return;

}

// On utilise notre BetaHRML

$response = $this->betaHTML->addBeta($event->getResponse(), $remainingDays);

// On met à jour la réponse avec la nouvelle valeur

$event->setResponse($response);

}

}

**Rappel :**

* **Tout d'abord, créez un service 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 une classe (le future listener) qui contient une méthode qui va faire le lien entre le déclenchement de l'évènement et le code que vous voulez exécuter (le service précédent). Il s'agit de la méthode processBeta 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 Symfony et les nôtres :

### Les évènements Symfony :

Symfony déclenche des évènements en interne, mais on peut créer les nôtres !

#### Kernel.request :

Est executé avant même que le choix du contrôleur ne soit fait. On peut le modifier pour y ajouter des attributs.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est GetResponseEvent, dont les méthodes sont les suivantes :

<?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 isMasterRequest();

public function isPropagationStopped();

public function stopPropagation();

}

#### Kernel.controller

Déclenché après l’analyse du contrôleur à exécuter ait été faite. Son objectif est de permettre à un Listener de modifier le contrôleur à exécuter.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est FilterControllerEvent, dont les méthodes sont les suivantes :

<?php

class FilterControllerEvent

{

public function getController();

public function setController($controller);

public function getKernel();

public function getRequest();

public function getRequestType();

public function isMasterRequest();

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 :

<?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);

}

#### 

#### Kernel.view :

Est déclenché lorsqu’un contrôleur n’a pas retourné d’objet Response. Son objectif est d’attraper la réponse du contrôleur s’il y en a une. Avec on peut construire une réponse ou personnaliser sa page d’erreur.

La classe de l'évènement donné en argument par le gestionnaire d'évènements est GetResponseForControllerResultEvent (rien que ça !), dont les méthodes sont les suivantes :

<?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 isMasterRequest();

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 :

<?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);

}

#### kernel.exception :

Est déclenché quand vient une exception. Son objectif est de permettre à un listener de modifier la réponse à renvoyer ou 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 :

<?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 isMasterRequest();

public function isPropagationStopped();

public function stopPropagation();

}

#### Security.interactive\_login :

Déclenché quand un utilisateur s’identifie sur le formulaire. Permet apr exemple d’archiver une trace de l’identification.

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 :

<?php

class InteractiveLoginEvent

{

public function getAuthenticationToken();

public function getRequest();

public function isPropagationStopped();

public function stopPropagation();

}

#### Security.authentification.success :

Déclenché quand on s’identifie avec succès. Permet aussi par exemple d’archiver une trace de connexion de l’utilisateur.

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 :

<?php

class AuthenticationEvent

{

public function getAuthenticationToken();

public function getRequest();

public function isPropagationStopped();

public function stopPropagation();

}

#### Security.authentification.failure :

Déclenché quand on échoue l’identification sur le formulaire.

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 :

<?php

class AuthenticationFailureEvent

{

public function getAuthenticationException();

public function getRequest();

public function isPropagationStopped();

public function stopPropagation();

}

### Créer nos propres évènements :

La logique d'un évènement est la suivante : j'ai une fonctionnalité, par exemple l'ajout de messages (d'une annonce, d'une candidature, etc.) et je souhaite pouvoir l'étendre de façon découplée. Sur l'ajout de message, on pourrait par exemple :

* déléguer l'indexation dans un moteur de recherche (pour ensuite faire des recherches textuelle),
* vérifier si le message est un spam (méthode différente de notre contrainte précédente),
* ou encore recevoir des notifications à chaque message posté, 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 ou ceux qui l'écoute(nt).
* 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 Platform. Mais en réalité, comme c'est une fonctionnalité qui s'appliquera à plusieurs bundles (la plateforme d'annonces, 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 :

On va déjà créer un fichier qui contient uniquement le nom des évènements. Facultative, mais bonne pratique.

:

<?php

// src/OC/PlatformBundle/Event/PlatformEvents.php

namespace OC\PlatformBundle\Event;

final class PlatformEvents

{

const POST\_MESSAGE = 'oc\_platform.post\_message';

// Vos autres évènements…

}

### Construire la classe de l’évènement :

La classe de l’évènement est la classe de l’objet que le gestionnaire d’évènement va transmettre aux listeners :

<?php

// src/OC/PlatformBundle/Event/MessagePostEvent.php

namespace OC\PlatformBundle\Event;

use Symfony\Component\EventDispatcher\Event;

class MessagePostEvent extends Event

{

}

Ensuite on rajoute la spécificité de notre évènement.

<?php

// src/OC/PlatformBundle/Event/MessagePostEvent.php

namespace OC\PlatformBundle\Event;

use Symfony\Component\EventDispatcher\Event;

use Symfony\Component\Security\Core\User\UserInterface;

class MessagePostEvent extends Event

{

protected $message;

protected $user;

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, les listeners ne peuvent pas modifier l'auteur du message !

}

### Déclencher l’évènement :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Event\PlatformEvents;

use OC\PlatformBundle\Event\MessagePostEvent;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// …

if ($form->handleRequest($request)->isValid()) {

// On crée l'évènement avec ses 2 arguments

$event = new *MessagePostEvent*($advert->getContent(), $advert->getUser());

// On déclenche l'évènement

$this->get('event\_dispatcher')->dispatch(*PlatformEvents*::POST\_MESSAGE, $event);

// On récupère ce qui a été modifié par le ou les listeners, ici le message

$advert->setContent($event->getMessage());

$em = $this->getDoctrine()->getManager();

$em->persist($advert);

$em->flush();

// …

}

}

}

Pour déclencher un évènement à proprement parler, il faut d'abord créer une instance dudit évènement, puis appeler la méthode dispatch de l'EventDispatcher :

* Le premier argument est le nom de l'évènement déclenché, ici la constance PlatformEvents::POST\_MESSAGE,
* Le deuxième argument est l'instance de l'évènement, ici notre instance $event.

### Ecouter l’évènement :

Voici UN exemple de Listener parmi d’autre :

<?php

// src/OC/PlatformBundle/Bigbrother/MessageNotificator.php

namespace OC\PlatformBundle\Bigbrother;

use Symfony\Component\Security\Core\User\UserInterface;

class MessageNotificator

{

protected $mailer;

public function \_\_construct(\Swift\_Mailer $mailer)

{

$this->mailer = $mailer;

}

// Méthode pour notifier par e-mail un administrateur

public function notifyByEmail($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);

}

}

Ainsi que le listener à proprement parler, qui vient exécuter cette fonction seulement lorsque l'auteur du message est dans une liste pré-définie (ici, je la passe en argument du constructeur) :

<?php

// src/OC/PlatformBundle/Bigbrother/MessageListener.php

namespace OC\PlatformBundle\Bigbrother;

use OC\PlatformBundle\Event\MessagePostEvent;

class MessageListener

{

protected $notificator;

protected $listUsers = array();

public function \_\_construct(MessageNotificator $notificator, $listUsers)

{

$this->notificator = $notificator;

$this->listUsers = $listUsers;

}

public function processMessage(MessagePostEvent $event)

{

// On active la surveillance si l'auteur du message est dans la liste

if (in\_array($event->getUser()->getUsername(), $this->listUsers)) {

// On envoie un e-mail à l'administrateur

$this->notificator->notifyByEmail($event->getMessage(), $event->getUser());

}

}

}

Et bien sûr, la définition des services correspondants :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.bigbrother.message\_notificator:

class: OC\PlatformBundle\Bigbrother\MessageNotificator

arguments:

- "@mailer"

oc\_platform.bigbrother.message\_listener:

class: OC\PlatformBundle\Bigbrother\MessageListener

arguments:

- "@oc\_platform.bigbrother.message\_notificator"

- ["alexandre", "marine", "pierre"]

tags:

- { name: kernel.event\_listener, event: oc\_platform.post\_message, method: processMessage }

## Aller plus loin :

### Les souscripteurs d’évènements :

Comme un Listener, il écoute, mais il peut écouter dynamiquement 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](http://api.symfony.com/2.1/Symfony/Component/EventDispatcher/EventSubscriberInterface.html), qui ne contient qu'une seule méthode : getSubscribedEvents(). Vous l'avez compris, cette méthode doit retourner les évènements que le souscripteur veut écouter.

Voici par exemple comment on pourrait transformer notre MessageListener en un souscripteur :

<?php

// src/OC/PlatformBundle/Bigbrother/CensorshipListener.php

namespace OC\PlatformBundle\Bigbrother;

use OC\PlatformBundle\Event\PlatformEvents;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class MessageListener 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(

*PlatformEvents*::POST\_MESSAGE => 'processMessage',

*PlatformEvents*::AUTRE\_EVENEMENT => 'autreMethode',

// ...

);

}

public function processMessage(MessagePostEvent $event)

{

// ...

}

public function autreMethode()

{

// ...

}

}

Bien sûr, il faut ensuite déclarer ce souscripteur au gestionnaire d'évènements. Pour cela, ce n'est plus le tag kernel.event\_listener qu'il faut utiliser, mais : kernel.event\_subscriber. Avec ce tag, le gestionnaire d'évènement récupère tous les souscripteurs d'évènements et les enregistre.

Pas besoin d'ajouter les attributs *event* et *method* sur le tag, car c'est la méthode getSubscribedEvents qui retourne ces informations :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.bigbrother.message\_listener:

class: OC\PlatformBundle\Bigbrother\MessageListener

arguments:

- "@oc\_platform.bigbrother.message\_notificator"

- ["alexandre", "marine", "pierre"]

tags:

- { name: kernel.event\_subscriber }

Avec priority, on peut définir l’ordre d’importance des évènements.

### La priorité des évènements :

Plus l’indice est élevé, plus il est exécuté tôt.

Vous pouvez la définir très simplement dans le tag de la définition du service. Voici comment la définir à 2 :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.beta.listener:

class: OC\PlatformBundle\Beta\BetaListener

arguments:

- "@oc\_platform.beta.html\_adder"

- "2017-06-01"

tags:

- { name: kernel.event\_listener, event: kernel.response, method: processBeta, priority: 2 }

Et pour les souscripteurs, voici comment adapter la méthode getSubscribedEvents pour y ajouter l'information de la priorité. Ici j'ai mis une priorité de 2 également :

<?php

// Dans un souscripteur :

static public function getSubscribedEvents()

{

return array(

*PlatformEvents*::POST\_MESSAGE => array('processMessage' => 2)

);

}

### La propagation d’évènement :

Tous les évènements qu’on a listé avaient comme méthode :  stopPropagation() et isPropagationStopped(). Si on stoppe, tous les évènements avec un indice plus bas pour cet évènement ne seront par exécutés.

Pour visualiser ce comportement, je vous propose de modifier légèrement notre BetaListener. Rajoutez cette ligne à la fin de sa méthode onKernelResponse :

<?php

// src/OC/PlatformBundle/Beta/BetaListener.php

namespace OC\PlatformBundle\Beta;

use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

class BetaListener

{

public function processBeta(FilterResponseEvent $event)

{

// ...

// On stoppe la propagation de l'évènement en cours (ici, kernel.response)

$event->stopPropagation();

}

}

#### 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 Symfony, et il est possible de créer les nôtres.
* Le code du cours tel qu'il doit être à ce stade est disponible sur la branche [iteration-18](https://github.com/winzou/mooc-symfony/tree/iteration-18) du dépot Github.

# Traduire son site

## Le principe :

Symfony va vraiment chercher une correspondance exacte de la chaîne à traduire. C'est pourquoi on ne parle pas de **dictionnaire**, mais de **catalogue**.

La locale utilisé par un site est souvent dans la session ou le coockie.

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 montants 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, etc.

**S’il n’y a pas de traduction ?**

1. Ce qui est déjà traduit dans la locale fr\_FR vous sera retourné ;
2. Ce qui n'est pas encore traduiten fr\_FR, 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.

## Configuration :

On définit notre langue locale :

# app/config/config.yml

parameters:

locale: fr # Mettez « fr » ici si ce n'est pas déjà le cas

On va ensuite utiliser ce paramètre locale dans la configuration :

# 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: { fallbacks: ["%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%

On crée enfin l’action :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

public function translationAction($name)

{

return $this->render('OCPlatformBundle:Advert:translation.html.twig', array(

'name' => $name

));

}

Et comme indiqué dans le contrôleur, il nous faut la vue traduction.html.twig, la voici :

{# src/OC/PlatformBundle/Resources/views/Advert/translation.html.twig #}

<html>

<body>

Hello {{ name }}!

</body>

</html>

## Bonjour tout le monde :

### Dire à Symfony de traduire :

La traduction est possible à deux endroits :

* Dans les contrôleurs (ou service)
* Dans la vue (la plus conseillé)

### Le filtre Twig :

**Un filtre, pour le langage Twig est déstinée à formater/modifier une valeur.**

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 :

<div>

<p>{{ message|trans }}</p>

<button>{{ 'cancel'|trans }}</button>

<button>{{ 'validate'|trans }}</button>

</div>

### La balise de bloc Twig :

Une autre possibilité de traduction depuis la vue consiste à encadrer tous les textes dans des blocs {% trans %} … {% endtrans %}

Mais on ne peut y placer **QUE DU TEXTE**, pas de balise ou autre.

<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>

### Le service translator :

Parfois, vous devrez malgré tout réaliser quelques traductions depuis un contrôleur ou un service, dans le cas d'inscriptions dans un fichier de log, par exemple. On fait donc appel à translator :

<?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');

Dans notre petit exemple, le filtre suffira :

{# src/OC/PlatformBundle/Resources/views/Advert/translation.html.twig #}

<html>

<body>

{{ 'Hello'|trans }} {{ name }}!

</body>

</html>

Mais rien n’est traduit ! On a la langue locale, la langue de déstination, mais on n’a pas de dictionnaire !

## Le catalogue :

C’est l’endroit où on va faire le lien entre les chaînes de caractères.

### Le format XLIFF

Quand on crée un Bundle avec la ligne de commande, on a dans ressource un fichier translation qui a été crée avec un exemple au format XLF :

<!-- src/OC/PlatformBundle/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 %}Symfony is great{% endtrans %}

ou

{{ 'Symfony is great'|trans }}

ou

$translator->trans('Symfony is great')) -->

<source>Symfony is great</source>

<!-- La chaîne cible, traduction de la chaîne ci-dessus -->

<target>J'aime Symfony</target>

</trans-unit>

</body>

</file>

</xliff>

Très verbeux, 20 lignes pour en traduire une.

### Format YAML :

Voici l'équivalent YAML du fichier messages.fr.xlf précédent :

# src/OC/PlatformBundle/Resources/translations/messages.fr.yml

# La syntaxe est : « la chaîne source: la chaîne cible »

Symfony is great: J'aime Symfony

Pour nos exemples on utilisera le format YAML, il faut donc supprimer message.fr.xlf

Si on a « : » il faudra mettre la source entre guillemet.

### Le format PHP :

La syntaxe est celle d'un simple tableau PHP, comme ceci :

<?php

// src/OC/PlatformBundle/Resources/translations/messages.fr.php

return array(

'Symfony is great' => 'J\'aime Symfony',

);

### Mise en cache du catalogue :

En dev’ le cache est généré à chaque excution mais pas en prod’ qui coup il faut vider le cache à la main pour voir les changements.

### Notre traduction :

Créez le fichier messages.fr.yml nécessaire pour traduire notre « Hello » en français. Vous devriez arriver au résultat suivant sans trop de problèmes :

# src/OC/PlatformBundle/Resources/translations/messages.fr.yml

Hello: Bonjour

### Ajouter un nouveau message à traduire :

On rajoute simplement à la ligne les nouveaux mots.

# src/OC/PlatformBundle/Resources/translations/messages.fr.yml

Hello: Bonjour

Bye: Au revoir

### Extraire les chaînes d’un site existant :

Il existe un outil qui permet d’extraire les chaînes entre les 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.

Mais ne fonctionne pas avec les traductions des contrôleurs.

La commande fonctionne comme doctrine :schema :update, on doit renseigner soit **–dump-message** soit **–force**.

* 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.xlf du bundle par exemple.
* La seconde option --force effectue la mise à jour des catalogues, tout en conservant une sauvegarde des versions précédentes.

Note : avec prefix= « … » on change le préfix par défaut qui est \_\_.

### Traduire une nouvelle ligne :

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 :

# src/OC/PlatformBundle/Resources/translations/messages.de.yml

Hello: Guten Tag

## Récupérer la locale de l’utilisateur :

Il y a 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.

### Routing et locale :

On doit ajouter un paramètre dans l’URL :

# app/config/routing\_dev.yml

oc\_platform\_translation:

path: /{\_locale}/traduction/{name}

defaults:

\_controller: "OCPlatformBundle:Advert:translation"

On peut tester en essayant [/fr/traduction/winzou](http://localhost/Symfony2/web/app_dev.php/fr/traduction/winzou) ou [/de/traduction/winzou](http://localhost/Symfony2/web/app_dev.php/de/traduction/winzou), le contenu de la page est bien traduit.

Pour appliquer notre local sur toutes nos routes, on revient sur le début du cours avec l’utilisation du prefixe :

# app/config/routing.yml

oc\_platform:

resource: "@OCPlatformBundle/Resources/config/routing.yml"

prefix: /{\_locale}/platform # Ici, on ajoute {\_locale} au préfixe !

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 faut cependant penser à bien définir les choix possibles, pas que quelqu’un essaye de traduire en une :

# app/config/routing.yml

oc\_platform:

resource: "@OCPlatformBundle/Resources/config/routing.yml"

prefix: /{\_locale}/platform

requirements:

\_locale: en|fr # les locales disponibles, séparées par des pipes « | »

On applique aussi des paramètres par défaut en cas de première connexion.

# 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: { fallbacks: [%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 :

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 :

# 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 :

{# Dans une vue #}

{% trans %}Le site où on apprend tout... à partir de zéro !{% endtrans %}

{# 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 :

# Dans un catalogue

site.devise: The website where you learn it all... from scratch!

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 :

{# Dans une vue #}

{% trans %}site.devise{% endtrans %}

{# ou #}

{{ 'site.devise'|trans }}

### Nicher les traductions :

Disponible uniquement au format YAML !

# Dans un catalogue

advert.edit.title: Édition d'une annonce

advert.edit.submit\_button: Valider

advert.show.edit\_button: Éditer l'annonce

advert.show.create\_button: Créer une nouvelle annonce

En factorisant, on obtient :

# Dans un catalogue

advert:

edit:

title: Édition d'une annonce

submit\_button: Valider

show:

edit\_button: Éditer l'annonce

create\_button: Créer une nouvelle annonce

Côté utilisation, dans les vues ou avec le service translator, rien ne change. Vous utilisez toujours {{ 'advert.edit.title'|trans }} par exemple.

### Retoure à la ligne :

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.

# 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.

premier\_paragraphe: >

OpenClassrooms 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'indendation en moins).

deuxieme\_paragraphe: |

Lorsque que vous vous connectez en tant que membre de OpenClassrooms 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.

### Utiliser des listes :

# Dans un catalogue

charte:

titre: Mentions légales

donnee:

# les éléments de liste sont précédés d'un tiret en YAML

- >

OpenClassrooms 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 de OpenClassrooms 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 :

{# Dans une vue #}

{% for i in 0..2 %}

<p>{{ ('charte.donnee.' ~ i )|trans }}</p>

{% endfor %}

### Utiliser les domaines :

Un domaine correspond à un fichier, il est interessant de regrouper par domaine, par exemple messages, on utilise donc message.XX.XX.

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 : $translator->trans('chaîne', array(), 'domaine').

### Domaines et Bundles :

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 special : validator :

Les messages d’erreurs sont traduits avec validator et non messages. C’est celui qui est utilisé apr défaut par exemple pour les messages des formulaires. On place donc nos messages dans validator.fr.yml

## Traduction dépendante de variables :

Exemple : En anglais on n’a pas d’espace entre un mot et ! alors qu’en français oui.

### Les placeholders :

# src/OC/PlatformBundle/Resources/translations/messages.fr.yml

hello: Bonjour %name% !

Et

# src/OC/PlatformBundle/Resources/translations/messages.en.yml

hello: Hello %name%!

Il faut donc adapter la vue :

{# src/OC/PlatformBundle/Resources/views/Advert/translation.html.twig #}

{{ 'hello'|trans({'%name%': name}) }}

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.

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 : $translator->trans('hello', array('%name%' => $name)).

Dans le cas d’un texte brute, sans modification, on utilise une valeur fixe :

{{ 'hello'|trans({'%name%': 'moi-même'}) }}

### Les placeholders dans le domaine validator :

Si on veut placer des nombres, on n’utilise pas les placeholders, mais une synthaxe spécifique avec value et limit :

# src/OC/PlatformBundle/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 }}"

Il est à noter aussi que les guillemets sont obligatoires ici.

### Gestion des pluriels :

Dans une syntaxe où on a : il y a (nombres) valeurs, on veut mettre un S si nombre>1 :

On utilise transchoice prévu par Symfony :

Le filtre :

{{ 'advert.nombre'|transchoice(nombre) }}

La balise :

{% transchoice nombre %}advert.nombre{% endtranschoice %}

Le service :

<?php

$translator->transchoice('advert.nombre', $nombre);

Le catalogue, quant à lui, contient donc les deux syntaxes dans une même chaîne source. Voici la syntaxe particulière à adopter :

# src/OC/PlatformBundle/Resources/translations/messages.fr.yml

advert:

nombre: "[0,1]Il y a %count% annonce|[2,+Inf]Il y a %count% annonces"

On voit donc deux choix séparés d’un pipe.

### Afficher les dates au format local :

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.phpdisponible 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.

On utilise localizedate à la place de date :

{{ 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.

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.

| **Format** | **Date** | **Heure** |
| --- | --- | --- |
| full | jeudi 15 novembre 2016 | 14:22:15 Heure normale de l’Europe centrale |
| long | 15 novembre 2016 | 14:22:15 HNEC |
| medium | 15 nov. 2016 | 14:22:15 |
| short | 15/11/16 | 14:22 |
| none | (rien) | (rien) |

On obtient donc :

Aujourd'hui nous sommes le {{ 'now'|localizeddate('full', 'none') }} et il est {{ 'now'|localizeddate('none', 'short') }}

L'extension ne se trouve pas dans le coeur de Twig, il vous faut donc d'abord ajouter la bibliothèque twig/extensions  dans votre composer.json , puis faire un composer update :

// composer.json

"require": {

// ...

"twig/extensions": "~1.3"

},

Puis, il faut activer l'extension, en rajoutant simplement cette définition de service dans votre fichier de configuration, cela fonctionne exactement comme l'extension Twig qu'on a faite il y a quelques chapitres :

# app/config/config.yml

# …

# Activation de l'extension Twig intl

services:

twig.extension.intl:

class: Twig\_Extensions\_Extension\_Intl

tags:

- { name: twig.extension }

### Récapitulatif :

Voici pour terminer un petit récapitulatif des différentes syntaxes complètes, sachant que la plupart des arguments sont facultatifs.

Les balises :

{# Texte simple #}

{% trans with {'%placeholder%': placeholderValue} from 'domaine' into locale %}maChaîne{% endtrans %}

{# Texte avec gestion de pluriels #}

{% transchoice count with {'%placeholder%': placeholderValue} from 'domaine' into locale %}maChaîne{% endtranschoice %}

Les filtres :

{# Texte simple #}

{{ 'maChaîne'|trans({'%placeholder%': placeholderValue}, 'domaine', locale) }}

{# Texte avec gestion de pluriels #}

{{ 'maChaîne'|transchoice (count, {'%placeholder%': placeholderValue}, 'domaine', locale) }}

Les méthodes du service :

<?php

$translator = $this->get('translator'); // depuis un contrôleur

// Texte simple

$translator->trans('maChaîne', array('%placeholder%' => $placeholderValue) , 'domaine', $locale);

// Texte avec gestion de pluriels

$translator->transchoice($count, 'maChaîne', array('%placeholder%' => $placeholderValue) , 'domaine', $locale)

Vous savez maintenant comment créer les traductions dans les différentes langues que vous souhaitez gérer sur votre site !

# Convertir les paramètres de requête

## Pourquoi un paramConverteur ?

### Récupérer des entités doctrine avant le contrôleur :

**Un paramConverteur convertit les paramètres de notre route au format que l’on préfère.**

Il agit après le routeur mais avant le contrôleur. Le résultat du paramConverteur est stocké dans les attributs de la requête, on peut donc les injecter dans les arguments de l’action du contrôleur.

### Un paramConverteur utile : DoctrineParamConverter :

**Convertit directement nos paramètres en entité Doctrine.**

Dans le contrôleur, au lieu de récupérer le paramètre de route {id}sous forme de variable $id, on va récupérer directement une entité Advert sous la forme d'une variable $advert, qui correspond à l'annonce portant l'id $id.

Il lève aussi l’exception si l’ID n’existe pas.

**Un paramConverteur est en fait un Listener qui écoute l’évènement kernel.controller.**

Rappel : il s’exécute quand on sait quel contrôleur appeler. Le paramConverter va lire la signature de la méthode du contrôleur pour déterminer type de variable que l’on souhaite. Grace à ça il crée un attribut de requête du même type.

Pour déterminer le type de variable, le paramConverter a deux solutions :

<?php

public function testAction(Advert $advert)

* Ici le type est Advert, il va donc créer une entité de la classe Advert.
* Ou alors on définit nous-même les paramètres @paramConverter

## Utilisation des paramConverters existants

### Utiliser le paramConverter Doctrine :

Appartient au Bundle Sensio/FramewordBundle acctivé par défaut.

### S’appuyer sur l’ID et le typage de l’agument :

C’est la plus simple et la plus utilisé, si on prend la méthode pour afficher une vue :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

requirements:

id: \d+

Le point important est que le paramètre s’appelle ID, c’est aussi le nom de l’attribut de l’entité.

Maintenant côté contrôleur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function viewAction(Advert $advert, $id)

{

// Ici, $advert est une instance de l'entité Advert, portant l'id $id

}

}

Maintenant Advert est une entité pleinement opérationnel, même plus besoin de if(null !== $advert). Il utilise le typage et non le nom ! on pourrait avoir $bidule en paramètre de fonction au lieu $id.

### Utiliser l’annotation pour faire correspondre la route et l’identité :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /advert/{advert\_id}

defaults:

\_controller: OCPlatformBundle:Advert:view

requirements:

advert\_id: \d+

Ici on a advert\_id et non id, donc on n’a pas de correspondance entre les attributs de Advert et le nom de notre paramètre. Il faut donc le spécifier dans le contrôeur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/\*\*

\* @ParamConverter("advert", options={"mapping": {"advert\_id": "id"}})

\*/

public function viewAction(Advert $advert)

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 variable que l'on veut injecter. Le advert de l'annotation correspond donc au $advert 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 advert\_id correspond à l'attribut id de l'Advert ».

On peut passer plusieurs correspondances d’arguments :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

// La route serait par exemple :

// /platform/advert/{advert\_id}/{skill\_id}

/\*\*

\* @ParamConverter("advertSkill", options={"mapping": {"advert\_id": "advert", "skill\_id": "skill"}})

\*/

public function viewAction(AdvertSkill $advertSkill)

### Utiliser les annotations sur plusieurs arguments :

On peut aussi passer plusieurs arguments dans l’URL :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /advert/{advert\_id}/applications/{application\_id}

defaults:

\_controller: OCPlatformBundle:Advert:view

Il est mieux dé définir à la main au lieu des devinettes pour plusieurs arguments.

<?php

/\*\*

\* @ParamConverter("advert", options={"mapping": {"advert\_id": "id"})

\* @ParamConverter("application", options={"mapping": {"application\_id": "id"})

\*/

public function viewAction(Advert $advert, Application $application)

### Utiliser le paramConverter Datetime :

Conterti une date d’un format défini en un objet de type Datetime :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_list:

path: /list/{date}

defaults:

\_controller: OCPlatformBundle:Advert:viewList

Et voici comment utiliser le convertisseur sur la méthode du contrôleur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

/\*\*

\* @ParamConverter("date", options={"format": "Y-m-d"})

\*/

public function viewListAction(\Datetime $date)

## Créer ses propres paramConverters :

### Comment sont exécutés les paramConverters ?

À l'origine de tout, il y a un listener, il s'agit de [Sensio\Bundle\FrameworkExtraBundle\EventListener\ParamConverterListener](https://github.com/sensio/SensioFrameworkExtraBundle/blob/master/EventListener/ParamConverterListener.php). Ce listener écoute l'évènement 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 :

<?php

foreach ($converters as $converter) {

if ($converter->supports($configuration)) {

if ($converter->apply($request, $configuration)) {

return;

}

}

}

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 Symfony trouve les convertisseurs ?

Pour les connaître, Symfony utilise **LES TAGS DES SERVICES**.

**Un convertisseur est avant tout un service sur lequel on applique le tag request.param\_converter.**

D’abords on crée un serviceà laquel on ajoute le tag request.param\_converter :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.paramconverter.json:

class: OC\PlatformBundle\ParamConverter\JsonParamConverter

tags:

- { name: request.param\_converter }

### Créer un convertisseur :

Créer la classe du paramConverteur.

**Un convertisseur doit implémenter l’interface ParamConverterInterface**

<?php

// src/OC/PlatformBundle/ParamConverter/JsonParamConverter.php

namespace OC\PlatformBundle\ParamConverter;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

use Sensio\Bundle\FrameworkExtraBundle\Request\ParamConverter\ParamConverterInterface;

use Symfony\Component\HttpFoundation\Request;

class JsonParamConverter implements **ParamConverterInterface**

{

function supports(ParamConverter $configuration)

{

}

function apply(Request $request, ParamConverter $configuration)

{

}

L’interface ne définit que deux méthodes : supports et apply.

### Support :

Doit retourner true quand le convertisseur souhaite convertir le paramètre, false sinon. Les informations sur le paramètre courant sont stockées dans $configuration, il contient :

* $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).

### Apply :

Doit 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 JsonparaConverter :

Imaginons que, d'une façon ou d'une autre, vous avez un paramètre de route qui contient un tableau en JSON, par exemple {"a":1,"b":2,"c":3}. On souhaite simplement transformer cette chaîne de caractères JSON en un tableau PHP, via la fonction json\_decode.

**La classe :**

<?php

// src/OC/PlatformBundle/ParamConverter/JsonParamConverter.php

namespace OC\PlatformBundle\ParamConverter;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

use Sensio\Bundle\FrameworkExtraBundle\Request\ParamConverter\ParamConverterInterface;

use Symfony\Component\HttpFoundation\Request;

class JsonParamConverter implements ParamConverterInterface

{

function supports(ParamConverter $configuration)

{

// Si le nom de l'argument du contrôleur n'est pas "json", on n'applique pas le convertisseur

if ('json' !== $configuration->getName()) {

return false;

}

return true;

}

function apply(Request $request, ParamConverter $configuration)

{

// On récupère la valeur actuelle de l'attribut

$json = $request->attributes->get('json');

// On effectue notre action : le décoder

$json = json\_decode($json, true);

// On met à jour la nouvelle valeur de l'attribut

$request->attributes->set('json', $json);

}

}

**Le contrôleur :**

Pour l’utiliser, on l’annote pour appliquer le convertisseur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

// N'oubliez pas le use pour l'annotation :

use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;

class AdvertController extends Controller

{

/\*\*

\* @ParamConverter("json")

\*/

public function ParamConverterAction($json)

{

return new *Response*(print\_r($json, true));

}

}

Un exemple de route :

# app/config/routing\_dev.yml

oc\_platform\_paramconverter:

path: /test/{json}

defaults:

\_controller: "OCPlatformBundle:Advert:ParamConverter"

Et accédez à la page [/test/{"a":1,"b":2,"c":3}](http://localhost/Symfony/web/app_dev.php/test/%7B%22a%22:1,%22b%22:2,%22c%22:3%7D). Le résultat est l'affichage par le fonction print\_r d'un tableau en pur PHP, qui a donc été décodé depuis le JSON initial.

# Personnaliser les pages d’erreurs

**Quand une exception est délcenché, le noyau la chope. Cela permet d’effectuer l’action adéquat.**

Le principe par défaut est d’appeler un contrôleur integré à Symfony : **TwingBundle :Exception :show**. Le contrôleur récupère les infos, choisit le template adéquat et envoie la réponse générée.

### Constater les pages d’erreur :

Il existe deux pages différentes : celle du dev et celle de la prod’.

La vue qui intéresse la plus est celle de la prod’ (visible par les utilisateurs).

### Localiser les vues concernées :

Elles se situent dans le Bundle TwigBundle vendor\symfony\symfony\src\Symfony\Bundle\TwigBundle\Resources\views\Exception .

### Remplacer les vues d’un Bundle :

Il suffit de créer app/Resources/NomDuBundle/views/ et d’y placer nos vues.

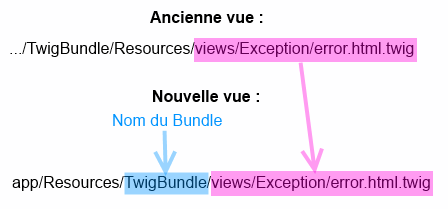
Nos vues doivent porter exactement les mêmes noms que celles qu'elles remplacent. Ainsi, si notre bundle utilise une vue située dans :

…(namespace)/RépertoireDuBundle/Resources/views/Hello/salut.html.twig

alors nous devons créer la vue :

app/Resources/NomDuBundle/views/Hello/salut.html.twig

Preuve en image :



### Comportement de Twig :

Twig regarde, pour chaque vue, s’il la trouve dans app/ressource, s’il ne la trouve pas, il va voir dans le 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.

**C’est le comportement de Twig ! Pas du reste**

### Pourquoi toutes les erreurs error.XXX.twig sont dans le répertoire Exception ?

Chaque vue d’erreur est compatible au format de son extension. Cela permet de générer des erreurs en cascade.

Je m'explique, imaginons que vous chargiez 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 Symfony fournit plusieurs formats d'erreur. Ainsi, si le format de votre réponse est défini comme du JavaScript, alors Symfony utilisera la vue error.js.twig qui est, si vous l'ouvrez, compatible JavaScript car en commentaire.

## Remplacer les templates exceptions du twig :

### Créer la nouvelle vue :

On va déjà créer le répertoire app/Resources/TwigBundle/views/Exception. Au sein du répertoire, le Bundle utilise la convention suivante :

* 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.

On va donc créer error404.html.twig.

Si on jette un coup d’œil à error.html on voit :

<!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>

Il se situe vendor\symfony\src\Symfony\Bundle\TwigBundle\Resources\views\Exception

On voit donc les codes que la page utilise avec status\_code et status\_text, on peut créer la notre maintenant.

# Utiliser Assetic pour gérer le CSS et JS

Souvent, ce qui fait ramer l’affichage de la page, c’est justement le CSS et le JS !

## Entre vitesse et lisibilité

Le temps de chargement comprend :

* 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 on a 10 JS, 3 JS et 15 images, ça fait 28 requêtes !

### Comment optimiser le front-end ?

Première idée, avoir un fichier CSS au lieu de plein de fichiers, comme ça on fait une seule requête CSS. Mais si on avait plein de fichiers, c’était pour la lisibilité, plus pratique et plus propre …

**Assetic va regrouper les fichiers en un seul. On pourra garder donc conserver notre séparation.**

### On peut aussi améliorer le temps de chargement :

Avec un seul fichier on a quand même plusieurs dizaines ou centaines de lignes … On va diminuer la taille du fichier grace à du JS : **YUI Compressor** un outil développé par Yahoo. Il supprime les espaces, commentaires, sauts de ligne.

<link href="//netdna.bootstrapcdn.com/bootstrap/3.0.2/css/bootstrap.min.css">

<link href="//netdna.bootstrapcdn.com/font-awesome/3.2.1/css/font-awesome.min.css">

<link href="/assets/css/style.css">

## La pratique :

### Installer Assetic :

Il faut l’installer avec ses dépendances : dans le fichier composer.json

"require": {

"symfony/assetic-bundle": "^2.7.1",

"leafo/scssphp": "~0.6",

"patchwork/jsqueeze": "~1.0"

},

Et faire un composer.phar update.

Et ajouter la configuration minimale du Bundle AsseticBundle dans app/config/config.yml :

# app/config/config.yml

assetic:

debug: '%kernel.debug%'

use\_controller: '%kernel.debug%'

### Servir des ressources :

#### Une seule ressource :

{# src/OC/CoreBundle/Resources/views/layout.html.twig #}

{# ou n'importe quelle vue en réalité ! #}

<link rel="stylesheet" href="{{ asset('bundles/ocplatform/css/main.css') }}" type="text/css" />

Et voici comment faire pour décharger cette responsabilité à Assetic :

{% stylesheets 'bundles/ocplatform/css/main.css' %}

<link rel="stylesheet" href="{{ asset\_url }}" type="text/css" />

{% endstylesheets %}

Il faut un fichier dans le répertoire : src/OC/PlatformBundle/Resources/public/css/main.css.

Attention, on utilise ici le fichier CSS déployé dans le répertoire web , et non celui qui se trouve dans le répertoire src/OC/PlatformBundle/... . Cela signifie que vous devez avoir publié vos assets au préalable, grâce à la commande php bin/console assets:install .

Pour éviter de devoir exécuter cette commande à chaque fois, je vous conseille de faire un lien (une sorte de raccourci), en ajoutant le paramètre assets:install --symlink . Ainsi, le dossier web/bundles/ocplatform  pointe en réalité vers src/OC/PlatformBundle/Resources/public , du coup lorsque vous changez l'un, vous changez l'autre ! Sous Windows, vous devez lancer l'invite de commande en mode Administrateur pour pouvoir le faire.

Et voici le HTML qu'Assetic a généré avec cette balise :

<link rel="stylesheet" href="/Symfony/web/app\_dev.php/css/519c4f6\_main\_1.css" type="text/css" />

Le principe est le même pour les fichier JavaScript :

{% javascripts 'bundles/ocplatform/js/main.js' %}

<script type="text/javascript" src="{{ asset\_url }}"></script>

{% endjavascripts %}

En cas d’erreur en mode dev, penser à vider le cache.

#### Servir plusieurs ressources regroupées en une :

On a rajouté une ligne de la même manière qu’avant :

{% stylesheets

'bundles/ocplatform/css/main.css'

'bundles/ocplatform/css/platform.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 :

<link rel="stylesheet" href="/Symfony/web/app\_dev.php/css/03b7e21\_main\_1.css" type="text/css" />

<link rel="stylesheet" href="/Symfony/web/app\_dev.php/css/03b7e21\_platform\_2.css" type="text/css" />

Ou alors charger tous ceux qui répertoire :

{% stylesheets 'bundles/ocplatform/css/\*' %}

Assetic a la possibilité de modifier tout ce qu’il sert grace aux filtres :

### Le filtre csswrite :

Si on a des images avec le chemin relatif : …/img/exemple.png, quand le fichier css était dans web/bundle/xxx/xxx, le chemin relatif pointait bien vers web/bundle/xxx/img, mais maintenant du côté du navigateur, le css est dans app\_dev.php/css.

On utilise donc csswrite :

{% stylesheets filter='cssrewrite'

'bundles/ocplatform/css/main.css'

'bundles/ocplatform/css/platform.css' %}

<link rel="stylesheet" href="{{ asset\_url }}" type="text/css" />

{% endstylesheets %}

Et activer aussi dans la configuration :

# app/config/config.yml

assetic:

debug: '%kernel.debug%'

use\_controller: '%kernel.debug%'

filters:

cssrewrite: ~

### Les ciltres scssphp et jsqueeze :

Sont installé plus haut grace à composer. Ils minifient le fichier.

Leur code se trouvent dans notre répertoire vendor. On va maintenant le configurer :

# app/config/config.yml

assetic:

debug: '%kernel.debug%'

use\_controller: '%kernel.debug%'

filters:

cssrewrite: ~

jsqueeze: ~

scssphp:

formatter: 'Leafo\ScssPhp\Formatter\Compressed'

Voilà, nous venons d'activer les filtres scssphp et jsqueeze, on peut maintenant les utiliser depuis nos vues. Ajoutez ce filtre dans vos balises :

{% stylesheets filter='cssrewrite, scssphp'

... %}

Et de même pour les fichiers JavaScript :

{% javascripts filter='jsqueeze'

... %}

En mettant un « ? » devant, on signal qu’on ne veut pas l’appliquer en mode DEV. Pratique pour pouvoir relire le fichier. Il serait illisible sinon !

{% stylesheets filter='?scssphp'

... %}

### Appliquer les chagements au mode prod’ :

**Comprendre Assetic :**

Quand on utilise les balises {{% stylesheet %}} (ou javascript), le code généré en HTML pour le prod est :

<link rel="stylesheet" href="/Symfony/web/css/cd91cad.css" type="text/css" />

En mode dev, Assetic passe directement par un contrôleur pour générer à la volée nos ressources. Mais minifier pour chaque requête prend du temps, qu’on ne peut se permettre en mode prod’.

**On va donc exporter en dur le fichier /css/cd91cad.css. Le serveur va directement envoyer le contenu au lieu de refaire toute la minification.**

### Exporter les fichiers CSS et JS :

On doit effectuer une commande pour pouvoir exporter :

php bin/console assetic:dump --env=prod

Cette commande devrait vous sortir un résultat de ce type :

C:\wamp\www\Symfony>php bin/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

Bien sûr cd91cad est un exemple, on peut avoir autre chose !

On peut aussi se référer à [la documentation officielle](https://symfony.com/doc/current/assetic.html).

# Utiliser la console directement depuis le navigateur

La console est un bon outil pour Symfony, mais ouvrir le terminale n’est pas tout le temps pratique.

## Le composant console de Symfony :

### Les commandes sont en PHP :

Toutes les fonctions que l’on a utilisé dans la console sont tous en PHP. On peut donc les exécuter depuis un script PHP, c’est ce qui se fait déjà ! Voici ce que contient bin/console :

#!/usr/bin/env php

<?php

use Symfony\Bundle\FrameworkBundle\Console\Application;

use Symfony\Component\Console\Input\ArgvInput;

use Symfony\Component\Debug\Debug;

set\_time\_limit(0);

$loader = require \_\_DIR\_\_.'/../app/autoload.php';

$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';

if ($debug) {

*Debug*::enable();

}

$kernel = new *AppKernel*($env, $debug);

$application = new *Application*($kernel);

$application->run($input);

### Exemple d’une commande :

#!/usr/bin/env php

<?php

use Symfony\Bundle\FrameworkBundle\Console\Application;

use Symfony\Component\Console\Input\ArgvInput;

use Symfony\Component\Debug\Debug;

set\_time\_limit(0);

$loader = require \_\_DIR\_\_.'/../app/autoload.php';

$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';

if ($debug) {

*Debug*::enable();

}

$kernel = new *AppKernel*($env, $debug);

$application = new *Application*($kernel);

$application->run($input);

### Exemple de commande :

Chaque commande est dans une classe PHP bien distincte, que l’on place dans le répertoire Command des Bundles. Ces classes comprennent (entre autre) 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 :

<?php

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

{

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 --xml</info>

EOF

);

}

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')));

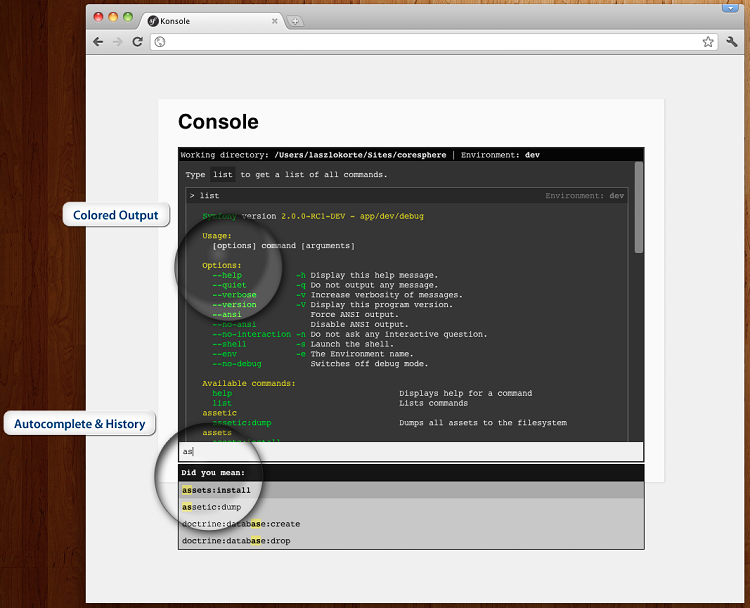
}

}

}

## Utiliser un console Bundle :

Il existe [plein de Bundle](http://knpbundles.com/ornicar/GravatarBundle) créés par la communauté ! Un qui est une référence pour la console dans le navigateur : CoreSphereConsoleBundle. C'est un bundle simple qui remplit parfaitement sa tâche, et dont l'interface est très pratique, comme le montre la figure suivante.



### Installer CoreSphereConsoleBundle :

* Télécharger le Bundle [sur GitHub](https://github.com/CoreSphere/ConsoleBundle).
* L’installer avec Composer.

Pour l'installer avec Composer, rajoutez la ligne suivante dans vos dépendances :

// composer.json

"require": {

// …

"coresphere/console-bundle": "~0.4"

}

Puis mettez à jour vos dépendances grâce à la commande suivante :

php ../composer.phar update

On l’enregistre ensutie dans le kernel :

Puis il faut enregistrer le bundle CoreSphereConsoleBundle dans app/AppKernel.php. Je ne l'active que pour le mode "dev" car on ne s'en servira que via le contrôleur frontal de développement (app\_dev.php) :

<?php

// app/AppKernel.php

use Symfony\Component\HttpKernel\Kernel;

use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel

{

public function registerBundles()

{

$bundles = array(

// ...

);

if (in\_array($this->getEnvironment(), array('dev', 'test'))) {

// ...

$bundles[] = new *CoreSphere*\ConsoleBundle\CoreSphereConsoleBundle();

}

return $bundles;

}

}

### Enregistrement des routes :

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 :

# app/config/routing\_dev.yml

coresphere\_console:

resource: .

type: extra

### Publier les Assets :

Il ne reste plus qu'à rendre disponibles les nouveaux fichiers JS et CSS du bundle, ce qui se fait grâce à la commande suivante :

php bin/console assets:install web

C'est fini ! Il ne reste plus qu'à utiliser notre nouvelle console.

### Utilisation de la console :

La route définie de base est \_console. On pourra y accéder et taper directement notre commande au lieu de faire bin/console.

# Déployer son site en production

La méthodologie est la suivante :

* Uploader sur le server de production
* Mettre à jour vos dépendances via Composer
* Mettre à jour la BDD.
* Vider le cahe

## Préparer l’appli en local :

D’abords on va vider le cache en dev et prod :

php bin/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 :

php bin/console cache:clear --env=prod

On va donc utiliser en app.php et non plus app\_dev.php. Ce qui n’est pas pratique c’est qu’on n’a aucune info en cas d’erreur. On va donc activer le debugger dans web/app :

<?php

// web/app.php

// …

$kernel = new *AppKernel*('prod', true); // Définissez ce 2e argument à true

Il faut bien sûr penser à le remettre à false quand on a fini les tests !

Quand le site est en prod sur le serveur, les erreurs sont logées dans var/logs/prod, on pourra y jetter un œil pour voir les erreurs rencontrées par les utilisateurs.

Soigner ses pages d’erreurs est une chose importante pour que le site reste professionnel. (CF chapitre dédié).

### Vérifier la qualité de son code :

Difficile de tout revoir si le code est immense. On utilise un outil en ligne : [insight SensioLab](https://insight.sensiolabs.com/?utm_source=openclassroom&utm_medium=parteners&utm_campaign=PARTENARIATS) pour vérifier que le code est bon. Pour fonctionner le projet doit être sur un GutHub public.

### Vérifier la sécurité des dépendances :

Pour vérifier la sécurité, comme pour la qualité, on utilise un outil en ligne [security.SensioLab](https://security.sensiolabs.org/?utm_source=openclassroom&utm_medium=security&utm_campaign=PARTENARIATS). On a deux façons de tester :

* Soit on envoie Composer.lock pour qu’il check les dépendances.
* Soit on utilise l’invité de commande

C:\wamp\www\Symfony>php bin/console security:check

Security Check Report

~~~~~~~~~~~~~~~~~~~~~

Checked file: C:\wamp\www\Symfony\composer.lock

[OK]

0 packages have known vulnerabilities

This checker can only detect vulnerabilities that are referenced

Disclaimer in the SensioLabs security advisories database. Execute this

command regularly to check the newly discovered vulnerabilities.

## Vérifier et préparer le serveur de production :

### Vérifier la compatibilité :

Deux cas :

**Un hébèrgeur :**

Symfony a un fichier qui fait le test de compatibilité dans web/config.php. ON doit le modifier :

<?php

// web/config.php

// …

if (!in\_array(@$\_SERVER['REMOTE\_ADDR'], array(

'127.0.0.1',

'::1',

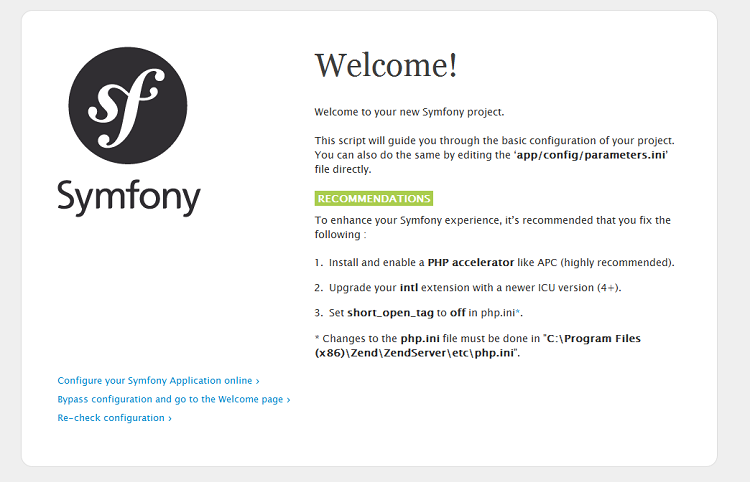
))) {

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](http://www.votre-serveur.com/config.php). Vous devriez obtenir la figure suivante :



Il n’y a pas la partie Major problems, juste les recommandations, donc le serveur est compatible. Il est aussi, important de suivre les recommandations !

**On n’a pas encore d’hébèrgeur :**

Voici les points à vérifier :

* La version de PHP doit être supérieure ou égale à PHP 5.5.9 ;
* 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](https://symfony.com/fr/doc/master/reference/requirements.html?utm_source=openclassroom&utm_medium=requirements&utm_campaign=PARTENARIATS).

## Déployer l’application :

Deux cas pour envoyer l’application :

* On n’a pas accès au SSH, on les envoie à la main.
* On a accès au SSH, on utilise alors Capifony pour automatiser.

### Méthode 1 : envoyer par FTP :

On supprime d’abords le cache pour gagner de la place. Si on a accès à Composer sur notre serveur, on n’envoie pas vendor mais juste les composer.lock et composer.json, on fera un php composer.phar install à distance.

Si on n’y a pas accès on envoie vendor.

#### Regler les droits sur /var :

Avec FileZilla, click droit sur var et aller dans les droits d’accès au fichier. On met 777(chmod).

### Méthode 2 : utiliser Capifony :

En cas de site plus volumineux, il est plus judicieux d’utiliser les bons outils : <http://capifony.org>. Devrait faire l’objet d’un cours à part entière. Se documenter personnellement.

## Les derniers préparatifs une fois le site en ligne :

Une fois le site en ligne, si on essaye d’accéder à app\_dev, accès interdit ! En effet il y a une reconnaissance d’IP à configurer :

<?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.');

}

### Mettre en place la BDD :

On va modifier app/config/parameters.yml pour que le serveur adapte les valeurs du paramètre 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 fichierparameters.yml, en exécutant cette commande :

php bin/console doctrine:database:create

Puis, dans tous les cas, remplissez la base de données avec les tables correspondant à vos entités :

php bin/console doctrine:schema:update --force

### Avoir de belles URL :

Pour ne pas rester avec des URL moches, on va se documenter sur [l'URL rewriting d'Apache](https://httpd.apache.org/docs/2.0/misc/rewriteguide.html).

L’objectif est que /platform et /css//style.css arrivent respectivement sur /web/platform et /web/css//style.css.

#### Méthode .htaccess :

Pour faire cela avec un.htaccess, rajoutez donc ces lignes dans un.htaccessà la racine de votre serveur :

<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/platform il faut qu'au final elle arrive sur/web/app.php/platform. En fait il y a déjà un.htaccessdans le répertoire/web. Ouvrez-le, il contient ce qu'il faut. Pour résumer, l'URL/platform va être réécrite en/web/platform par notre.htaccessà la racine, puis être à nouveau réécrite en/web/app.php/platform par le.htaccessde Symfony situé dans le répertoire/web.

#### Méthode virtualHost :

Si on a accès au serveur http, c’est la meilleure méthode. L’objectif est de modifier <http://localhost/Symfony> en <http://Symfony>.

# 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>

votre propre PC, soit localhost (ou encore 127.0.0.1). Pour cela il faut modifier le fichierhosts, que vous trouverez ici :C:\Windows\System32\Drivers\etc\hosts. Rajoutez simplement une ligne avec :127.0.0.1 symfony.local. Ainsi, quand vous tapezhttp://symfony.localdans 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.

### Mise à jour de la BDD :

Perdre des données lors d’une MAJ est impenssable pour les utilisateurs …

C'est pour cela que la commande doctrine:schema:update est à bannir. Par contre, il existe une bibliothèque pour Doctrine appelée [DoctrineMigration](http://docs.doctrine-project.org/projects/doctrine-migrations/en/latest/index.html) et bien sûr [son bundle](https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html) correspondant. Je ne détaillerai pas son utilisation ici, mais l'idée est la suivante :

* Lorsque vous modifiez ou ajoutez une entité, vous créez en même temps un fichier de migration ;
* Le fichier de migration reflète les changements que vous avez effectués : il contient les requêtes SQL (du SQL pur, pas du DQL !) permettant de mettre à jour la base de données pour coller à vos changements. Il contient les deux sens : pour mettre à jour (ajout d'une colonne par exemple), et pour annuler la mise à jour (suppression de la nouvelle colonne) ;
* Lorsque vous envoyez vos fichiers sur votre serveur de production, vous exécutez alors tous les fichiers de migrations depuis la dernière mise en production : votre base de données sur le serveur est alors bien synchronisée avec votre code !

Pour être sûr qu’il ne manque RIEN, on utilise ce site : [checklist](http://www.symfony2-checklist.com/fr?utm_source=openclassroom&utm_medium=checklist&utm_campaign=PARTENARIATS) qui permet de vérifier que l’on a rien oublié.