Sécurité et généralités

* Chaîne de requête : partie d’une URL commençant par un point d’interrogation ( ?) qui contient des informations arbitraires sous la forme nom=valeur.
* Injection SQL : permet d'injecter dans la requête SQL en cours un morceau de requête non prévu par le système et pouvant en compromettre la sécurité.
* CSRF : Cross-Site Request Forgery, L’objet de cette attaque est de transmettre à un utilisateur authentifié une requête HTTP falsifiée qui pointe sur une action interne au site, afin qu'il l'exécute sans en avoir conscience et en utilisant ses propres droits. L’utilisateur devient donc complice d’une attaque sans même s'en rendre compte.
* XSS : cross-site scripting, d'injecter du contenu dans une page, permettant ainsi de provoquer des actions sur les navigateurs web visitant la page. Il est par exemple possible de rediriger vers un autre site pour de l'hameçonnage ou encore de voler la session en récupérant les cookies.
* Paramètre : donnée manipulée par une section de code (fonction, méthode) et connue du code appelant cette section.
* .htaccess : Permet de faire le lien avec le serveur Apache (configuration, redirection d'URL, filtre d'IP...)

POO

* POO : faire de son site un ensemble d'objets qui interagissent entre eux
* Représentation procédurale : consiste à séparer le traitement des données des données elles-mêmes
* Classes : Un modèle / plan de fabrication qui va contenir des propriétés ou attribut (variables) et des méthodes (fonctions) d'un objet
* Objet : Une variable qui a été définit à partir d'une classe qui contient des propriété et des comportements (méthodes). C'est une instance :
  + Instance : c'est tout simplement le résultat d'une *instanciation*
  + Instanciation : c'est le fait d'*instancier* une classe
  + Instancier une classe : se servir d'une classe afin qu'elle nous crée un objet
* Attribut : désigne une variable
* Méthodes : désigne une fonction / comportement en POO
* Encapsulation : Permet de protéger les données extérieures
* Visibilité : accessibilité de la classe : private, public, protected
* Public : appeler des attributs ou méthodes depuis l'extérieur de l'objet
* Private : accès aux attributs et méthodes *seulement* depuis l'intérieur de la classe
* Protected : Propriété / attributs accessible uniquement dans la classe en question ET dans les classe filles (celles qui en héritent)
* Classe Abstraite : Pas possible de l’instancier, on peut définir des méthodes abstraites qui seront définies dans les classe filles. Utilisation du mot clé »abstract .
* Méthode abstraite : syntax = *public abstract function nom() ;* (Pas d'accolade, car définiti déjà dans la classe abstraite)
* Méthode : syntax = *public function nom() {…} ;*
* Composition : Une classe en utilise une autre
* Héritage : concerne les classes (et leur hiérarchie), Utilisation du Extends
* Extends : Permet de dire qu'une classe Fille et lié à la classe mère.
* Polymorphisme : relatif aux méthodes des objets.
  + Le [polymorphisme ad hoc](http://www.commentcamarche.net/contents/" \l "adhoc) (également surcharge ou en anglais overloading) : permet d'avoir des fonctions de même nom, avec des   
    fonctionnalités similaires, dans des classes sans aucun rapport entre elles
  + Le [polymorphisme paramétrique](http://www.commentcamarche.net/contents/" \l "parametrique) (également généricité ou en anglais template) : possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type).
  + Le [polymorphisme d'héritage](http://www.commentcamarche.net/contents/" \l "heritage) (également redéfinition, spécialisation ou en anglais overriding) : La possibilité de redéfinir une méthode dans des classes héritant d'une classe de base s'appelle la spécialisation. Il est alors possible d'appeler la méthode d'un objet sans se soucier de son type intrinsèque
* ?NameSpace :
* Autoloder : Charge automatique toutes les classes, évite de faire des <require>. Dans le fichier composer.json, après l'avoir installé, on doit mettre

"require": {},

"autoload": {

"psr-4":{

"": "src/" // Les guillement sont les namespaces vides

* Vendor : Bibilotheque externe
* $this : La pseudo-variable $this est disponible lorsqu'une méthode est appelée depuis un contexte objet. $this est une référence à l'objet appelant (habituellement, l'objet auquel la méthode appartient, mais ce peut être un autre objet si la méthode est appelée de manière statique depuis le contexte d'un autre objet).
* Instancier : $form = new Form() ;
* Constructeur : Méthode qui va être appelée au moment ou j'instancie ma classe. Syntax =

class Magicien extends Personnage

public function \_\_construct()

{

$this->force = 20 :

$this->pv = 50 ;

}

* Setteur : pour changer la propriété
* Getteur : Pour récupérer la valeur qui a été changer dans le setteur
* -> : permet à un objet instancier (ou une instance) d'accéder à des propriété ou des méthodes, accèder à qqch qui appartient à un objet

MVC

* Modèle : cœur (algorithmique) de l'application : traitements des données, interactions avec la base de données
* Vue : n'effectue pas de traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur
* Contrôleur : ne s’occupe que du code de gestion du protocole http (chaine de requete, formulaire, redirection http, session, cookie, authentification

HTML

* Canvas : une surface de dessin bitmap manipulable en JavaScript spécifique à HTML. Permet de dessiner animation, jeux, graphiques…
* SVG : format de fichier pour le dessin vectoriel, décrit en XML. Langage de balise. Permet de dessiner animation, jeux, graphiques…
* DOM : structure d'une page HTML

CSS

* :nth-child(n) (ex : p:nth-child(2) ) : Selects every <p> element that is the second child of its parent
* Pseudo-classe : un mot-clé qui peut être ajouté aux sélecteurs afin d'indiquer l'état spécifique dans lequel l'élément doit être pour être ciblé par la déclaration
* Sélecteurs CSS : définissent les éléments sur lesquelles s'applique un ensemble de règles CSS
* ? Background-position : permet de définir la position initiale, relative l'origine définie par background-origin pour chaque image d'arrière-plan.
* Reset : technique qui consiste à réinitialiser à 0 la valeur de certains éléments HTML afin d'éviter une partie des différences d'affichage sur les divers navigateurs
* Normalize : S’arrure que toutes les marges de tous les éléments aient la même valeurs dans tous les navigateurs

Javascript

* Paramètre : variable nommée passée à une fonction. Les paramètres servent à importer des arguments à l'intérieur des fonctions
* Argument : données traitées par une fonction. Ils sont remplacés par les entrées lors de l'exécution du programme qui utilise cette fonction.
* ? Prototype :
* HTMLElement.dataset : permet l'accès, à la fois en mode lecture et écriture, à tous les attributs de données sur mesure (data-\*) définis sur l'élément
* This : Opérateur. Dans le contexte global d'exécution (c'est-à-dire, celui en dehors de toute fonction), this fait référence à l'objet global (qu'on utilise ou non le mode strict).
* Bind() : Fonction qui a pour contexte this la valeur passée en paramètre et éventuellement une suite d'arguments qui précéderont ceux fournis à l'appel de la fonction créée.
* Initialiser un tableau : var tab = [] ;
* indexOf() : renvoie le premier indice pour lequel on trouve un élément donné dans un tableau
* Méthode : fonction associée à un objet, c'est-à-dire une action que l'on peut faire exécuter à un objet
* Méthode de JQUERRY : .on() (équivaklent à document.addEventListener en natif JS), .hide()
* fonction : function somme(a, b, c) {var d = a + b + c; return d;}var total = somme(2, 3, 5);
* XMLHttpRequest = Objet permettant de faire des requete AJAX en natif
* Requetes AJAX avec JQUERY :
  + $.get
  + $.post
  + $.ajax
  + $.load

PHP

* Superglobale : variables internes qui sont toujours disponibles, quel que soit le contexte
* $\_GET : Superglobale, tableau associatif des valeurs passées au script courant via les paramètres d'URL
* $\_POST : Superglobale, tableau associatif des valeurs passées au script courant via le protocole HTTP et la méthode POST lors de l'utilisation de la chaîne application/x-www-form-urlencoded ou multipart/form-data comme en-tête HTTP Content-Type dans la requête.
  + Pour récupérer une valeur saisie, il faut récupérer dans un tableau associatif, l’attribut « name » : $\_GET[‘name’];
* htmlentities() : Convertit tous les caractères éligibles en entités HTML
* Structure de contrôle : Conditions, boucles, Switch, if, for…
* FOR :
  + Initialisation : var i = 100 ou 0
  + Condition : i > 10 ;
  + Incrémentation / Décrémentation: i-- ou i++
* WHILE : le code s'exécute tant que la condition est vraie
* Tableau : syntax identique
  + ["td", 5, true] ;
  + array("td", 5, true) ;

SQL

* Fonction d’agrégation :
  + AVG() : pour calculer la moyenne sur un ensemble d’enregistrement
  + COUNT() : pour compter le nombre d’enregistrement sur une table ou une colonne distincte
  + MAX() : pour récupérer la valeur maximum d’une colonne sur un ensemble de ligne. Cela s’applique à la fois pour des données numériques ou alphanumérique
  + MIN() : pour récupérer la valeur minimum de la même manière que MAX()
  + SUM() : pour calculer la somme sur un ensemble d’enregistrement
* Sérialiser : extraire les données d'une instance en vue de la réhydrater plus tard (à un prochain lancement ou lorsque les données se sont rendues sur un autre poste, par exemple). Sérialiser permet aussi d'échanger des données entre plusieurs langages. Possible en XML et en JSON

Installer Silex

1. créer un fichier .htaccess :
   1. Terminal : touch .htaccess ENTER
   2. Terminal : gedit .htaccess ENTER
   3. Cela ouvre un fichier, copier coller cela

*<IfModule mod\_rewrite.c>*

*Options -MultiViews*

*RewriteEngine On*

*#RewriteBase /path/to/app*

*RewriteCond %{REQUEST\_FILENAME} !-d*

*RewriteCond %{REQUEST\_FILENAME} !-f*

*RewriteRule ^ index.php [QSA,L]*

*</IfModule>*

1. Installer le moteur de template Twig : composer require twig/twig
2. Une fois installer : se rendre sur : <http://silex.sensiolabs.org/doc/master/providers/twig.html> et copier coller / Registering
3. Créer ensuite un dossier Views (équivalent des .phtml)
4. Créer un fichier layout.html.twitg

TERMINAL

* ls : liste
* cd : se déplacer
* cd .. : revenir dans le dossier parent
* cd .. /../ : revenir plus en haut dans l'arbo
* php bin/console : affiche toutes les lignes de commandes de la console
* php console debug:router : permet de voir toutes les routes
* php composer.phar init : permet de configurer composer et de créer le fichier composer.json
* composer.json : ?
* php composer.phar (ou ici « composer » a la place de php 'composer.php') dump-autoload -o  : Genere automatiquement les fichiers d'autorisation de fichiers avec le -o
* ls -al : afficher les fichiers (y compris caché) avec '-al' sous forme de liste / tableau
* php bin/console cache:clear : permet de cleaner l'environnement de dev
* php bin/console cache:clear --env=prod : permet de cleaner l'environnement de prod

Faire le lien avec une base de donnée

* php bin/console doctrine:generate:entity : créer la class, permet de créer le fichier Entitiy permet de faire le lien entre le code et la base de données. Les entités sont liées à des tables. On peut en créé plusieurs et après on fait le schema :update
* php bin/console doctrine:schema:update : permet de vérifier la BDD
* Avant, créer la base de donnée sur PhpMyAdmin
* Puis créer la table
* appuyer sur les fleche pour sélectionner le bon Bundle, puis Tab et mettre après  « AdminBundle » pour que ca fasse « AdminBundle:<nom de la table>
* Les id sont automatiquement généré
* Ensuite répondre aux questions
* nullable : est-ce que le champs peut etre vide
* Cela va créer 2 dossiers Entity et repositorry
* Pui faire le lien avec PHPmyadmin en rentrant : php bin/console doctrine:schema:update
* Soit avec –force pour forecer 'lexecution ou sinon –dump-sql pour voir le codequ'il va faire
* php bin/console doctrine:generate:form AdminBundle:Product : Création du formulaire depuis le terminal pôur faire le lien (à faire pour que ca s'affiche)
* Documentation de doctrine : <http://www.doctrine-project.org/api/orm/2.5/class-Doctrine.ORM.EntityRepository.html>
* findBy et findONeBy

Générer un CRUD automatiquement

* Créer une entitée : : php bin/console doctrine:generate:entity, on rentre le nom de l'entité
* On met à jour le schéma avec : php bin/console doctrine:schema:update –force
* Généréer le crud doctrine:generate:crud

Générer des getter / setter

* php bin/console doctrine:generate:entities AdminBundle:pour mettre à jour / créer sur tout le dossier AdminBundle
* php bin/console doctrine:generate:entities AdminBundle:Category : si on veut spécifier sur un fichier précis
* Et mettre à jour la BDD avec : php bin/console doctrine:schema:update --force

Architecture

* GIT: Versionning
* Vendor : pour les dépendances. Ne pas s'amuser avec
* composer.json : ?
* loc : bloque la version qui a été prise à un moment prévis
* DefaultControllerapp : configuration générale de l'application
  + parameters.xml : c'est là qu'on peut changer les identifiants de connexion
* bin : regroupe les fichiers des exécutables. Permet de générer des controllers, vue, classe, mettre à jour la BDD
* SRC : sources du sites, dans lequel il y un fichier controller
* TDD : test driving developpement
* var : fichiers généré automatiquement
* log : journal qui génére des element (erreurs notamment)
* cache : évite de réinterpréter des traitements
* web : porte d'entrée du site. Il doit y avoir un fichier index.php
* htaccess : configuration et donne des règles particulières
* bundle : fonctionnalités
* app Kernel : fichier générale qui récupère des dépendances
* prefixes : dans le fichier routing.ymlk, mettre le nom qu'on souhait pour définir les routes dans la partie « prefix ». Une fois fait, aller dans terminal et faire *php bin/console debug:router* et voir quelles sont les routes qui bug et à changer dans les controllers, dans @route

Syntaxe et code TWIG

* {{ }} : permet d'afficher, comme un echo en php
* {# #}: commentaire
* { %%} : traitement
* <a href="{{ path('homepage') }}"> : permet de faire un lien avec PATH('nom-de-la-page'). ON pourrait remplacer PATH par URL, qui permet de faire des liens absolues, dans le cas d'une newsletter par exemple
* {{ 'WELCOME'|lower }} : Filter, avec le l qui permet d'exécuter un traitement
* Filters : ? je modifie la sortie de la varibale
* Fonctions : un traiement sur ce tableau
* DUMP : permet de débuguer en twig : dump($user) ;
* {{ asset(''dist/css/AdminLTE.min.css'') }} : fais le lien dans le href pour afficher le css
* {% include 'AdminBundle:Partials:footer.html.twig' %} : pour intégrer un élément qui est dans un autre fichier, et dans un autre fichier Partials
* asset : postionne dans un dossier web
* with : propre à include, pour faire passer des variable à la vue

{% include 'Partials/footer.html.twig' with

{

'firstname' : firstname,

'foo' : 'bar'

} only %}

* only : permet de ne pas faire passer des variable à la vue (à confirmer)
* {{ form(formContact) }} : affiche le formulaire, depuis des ressources stockées dans Symfony. FormContact étant une variable passée dans le tableau
* {{ form\_start(formContact) }} et {{ form\_end(formContact) }} : permet d'ouvrir le formulaire afin de mettre des {{ form\_row(formContact) }}, ppour custom
* {{ form\_row(formContact) }} : permet de customizer le formulaire, avec {{ form\_label(formContact.email) }}, {{ form\_widget(formContact.email) }} (qui correspond à input) et {{ form\_errors(formContact.email) }} (permet d'afficher des erreurs, par défaut UL et LI)

Syntaxe et code POO

* use : permet d'appeller plusieurs fois une class
* expression régulère : http://www.cbs.dtu.dk/courses/27610/regular-expressions-cheat-sheet-v2.pdf

GIT

* Outil GIT : Permet de versionner, état à un moment précis pour sauvegarder son travail
* Pour l'installer : <https://git-scm.com/downloads>
* git init : terminal, permet de créer un dossier .git (qui est caché)
* git status : suivi du projet. Terminal
* git add <fichier> : permet d'ajouter un fichier Terminal
* .gitignore : dans ce fichier, on peut ignorer autant de chose qu'on souhait, en rentrant la liste des éléments, des fichiers… Terminal
* git add . : Avec le '. ' permet de tout sélectionner de d'ajouter Terminal
* git commit -m '' message '' : permet de créer la sauvegarde avec un message permettant de savoir où on en est. Terminal
* Master : tronc commun du projet
* Branche : permet de faire des tests
* git log : permet de voir ce qu'il y a dans le git
* remote : permet d'envoyer à l'endroit qu'on veut
* git remote add origin <https://github.com/AlexisPG/poo.git>  : permet d'ajouter un projet sur l'URL
* git push -u origin master : permet de push un projet pour le sauvegarder
* git clone <url dans GitHub> : permet de cloner le projet dans un dossier
* git pull <fichier> master : permet de récupérer le dossier depuis git\*

Sauvegarde sur GIT :

* composer update
* git init
* git add .
* git config --global user.name "arphenn"
* git commit -m "NOM VERSION"
* git remote add origin https://github.com/AlexisPG/Projet-BO-et-FO
* git push -u origin master

Formulaire et envoyer un email

Dans le controller :

*/\*\**

*\** ***@Route****("/contact", name="contact")*

*\* Liste des éléments pour les formulaire :* ***https://symfony.com/doc/current/reference/forms/types.html***

*\*/*

**public function** contactAction(Request $request)

{

// Création du formulaire et ajoute de champ avec la méthode add()

$formContact = $this->createFormBuilder()

->add('firstname', TextType::*class*)

->add('lastname', TextType::*class*)

->add('email', EmailType::*class*)

->add('content', TextareaType::*class*)

->getForm();

// Je lie l'objet $request avec le formulaire.

// Cela me permet de remplir le formulaire avec les informations tapées par l'utilisateur

$formContact->handleRequest($request);

// Je vérifie que le formulaire est bien soumis et qu'il est valide

**if** ($formContact->isSubmitted() && $formContact->isValid()) {

// Dump de $\_POST

dump($request->request->all());

// Dump de $\_POST['firstname']

dump($request->request->get('firstname'));

// Dump de $\_GET

dump($request->query->all());

// Récupérer les informations du formulaire

dump($formContact->getData());

// Récupérer une valeur précisément du formulaire

dump($formContact->get('firstname')->getData());

// La technique à utiliser est d'utiliser une variable ex: $data et de manipuler cette variable

$data = $formContact->getData();

// Envoie du mail

// Affichage d'un message de succes

$this->addFlash('success', 'Votre email a bien été envoyé');

//Redirection : Préciser le nom de la route dans la méthode 'redirectToRoute

**return** $this->redirectToRoute('contact');

}

**return** $this->render('Default/contact.html.twig',

[

"formContact" => $formContact->createView()

]);

}

Dans la view (en twig)

{% **block** content %}

{# Affichage du message de success #}

{% **for** flash\_message **in** app.session.flashBag.get('success') %}

<div class="alert alert-success">{#Permet de mettre le message en vert, au niveau de la class#}

{{ flash\_message }}

</div>

{% **endfor** %}

{#Création du formulaire de cointact depuis le cotnroller DefaultController.php #}

{{ form\_start(formContact) }}

<div class="form-group">

{# affiche les erreurs du champ firstname #}

{{ form\_errors(formContact.firstname) }}

{# affiche la balise label du champ firtsname #}

{{ form\_label(formContact.firstname) }}

{# affiche la balise input de type text du champ firtsname #}

{{ form\_widget(formContact.firstname) }}

</div>

<div class="form-group">

{{ form\_errors(formContact.lastname) }}

{{ form\_label(formContact.lastname, "Votre nom", {label\_attr: {class: "le-nom-de-votre-class"} } ) }}

{{ form\_widget(formContact.lastname, {attr: {placeholder: "Votre nom", class: "form-control"} } ) }}

</div>

<div class="form-group">

{{ form\_errors(formContact.email) }}

{{ form\_label(formContact.email, null) }}

{{ form\_widget(formContact.email,

{

attr: {

placeholder: 'email',

class: "form-control"

},

id: "votre-super-id"

})

}}

</div>

<div class="form-group">

{{ form\_errors(formContact.content) }}

{{ form\_label(formContact.content) }}

{{ form\_widget(formContact.content) }}

</div>

<button>Envoyer</button>

{{ form\_end(formContact) }}

{% **endblock** content %}

Utiliser MalTrap pour faire des envois de test, changer utiliser Symfony dans le select, changer les parameters et mettre à jour les bonnes clés dans le fichier parameters.yml.dist

* Pour les restrictions : Ne pas oublier de mettre un USE avec un AS et mettre Assert pour pouvoir les utiliser dans les new Assert/<nom de la restriction> et mettre un 'constraint' => et les info, avec un tableau, si on en pleins <https://symfony.com/doc/current/reference/constraints.html>
* Pour les restrictions d'email : <https://symfony.com/doc/current/reference/constraints/Email.html>
* new Assert\Regex : permet de limiter les expression. '/\b(hello|je)\b/' permet de limiter uniquement le mot, et pas le fait que l'expression soit dans un mot

Pour modifier le design des formulaire ou autres

* Aller dans vendor/symfony/symfony/src/symfony/bridge/twig/Ressources/views/Form > Aller dans le fichier mais ne PAS MODIFIER, car si mise à jour, ça va buger
* 3 technique, celle la est la 2ème : Pour appeler, aller dans app/confi/config.yml, indiquer la page qui contient ce qu'on veut modifier, avec une indentation, puis créer la page twig, puis prendre le contenu du {{ block }} à modifier, et modifier dans le twig <https://symfony.com/doc/current/form/form_customization.html>

Pour modifier les erreurs, page 404, 500 …

* aller  dans vendor/symfony/symfony/src/symfony/Bundle/TwigBundle/Ressources/views/Exception, et regarder le fichier error.html.twig
* Puis créer dans app un nouveau chemin qui ira taper ici, plutot que dans error, en faisant le chemin : app/Ressources/TwigBundle/views/Exceptions

Evenements sur le formulaire

Documentation : <https://symfony.com/doc/current/form/events.html>

* Evenemnts utilisable pour agir sur le formulaire:
  + PRE\_SET\_DATA
  + POST\_SET\_DATA : a ce moment qu'on peut créer des donnée / gérer l'enregistrement des données
  + PRE\_SUBMIT
  + SUBMIT
  + POST\_SUBMIT
* Souscripteur : class qui permet d'écouter plusieurs évenements
* Inteface : class model qui doit être implémentée par des méthodes. En fait le « implements \Interface », pour connaître les méthodes → Clic droit → generate → Implements Methods
* Pour le souscripoteur soit actif il faut :
  + gérer services.yml
    - Vogeir la doc ici : http://symfony.com/doc/current/reference/dic\_tags.html
  + class de formulaire

Contraintes et restriction des entities (pour la base de donnée / édition)

* aller dans src/AdminBundle/Entity et mettre les contrainte dans les commentairs (annotation)
* Pour que toutes les entités sont obligatoires

/\*\*

\* @Assert\NotBlank()

\*/

Param converter

* Documentation : http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html
* Si il y a plus de variable / paramètres : [http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html#doctrineconverter-options](http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html" \l "doctrineconverter-options)

Stopper un evenement en JS (le bouton supprimer par exemple)

<script>

/\* Permet d'afficher la pop up etes vous sur de supprimer ? en prennat en compte l'élément parent d'un élemtn cré ou mùeme créé dynamiquement en AJX\*/

$(document).ready(**function**()

{

**var** table = $('.table');

table.on("click", '.btn-danger', **function**(event)

{

//console.log($(this))

event.preventDefault();

**var** elementA = $(**this**);

**var** elementParentTable = elementA.closest('.table');

**var** linkUrl = elementA.attr("href");

**if** (confirm("Etes-vous sûr de vouloir supprimer ce produit ?"))

{

// Requete Ajax

// http://api.jquery.com/jquery.ajax/

$.ajax({

url: linkUrl,

method: "GET"

}).done(**function**(data){

elementA.closest('tr').fadeOut(700, **function**(){

$(**this**).remove();

// Suppression des éventuels messages existants

$('.alert-success').remove();

// Ajout du message

**var** message = $('<p class="alert alert-success">').append(data.message);

elementParentTable.before(message);

message.delay(2000).fadeOut()

})

});

}

})

});

</script>

Afficher toutes les catégories

Dans la vue :

{{ render(controller("AdminBundle:Category:renderCategories")) }}

dans le twig :

{% **for** cat **in** categories %}

<li><a href=""><i class="fa fa-circle-o"></i> {{ cat.title }}</a></li>

{% **endfor** %}

dans le controller :

**public function** renderCategoriesAction() {

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

$categories = $em->getRepository('AdminBundle:Category')->findAll();

/\* die(dump($categories));\*/

**return** $this->render('Category/renderCategories.html.twig', ['categories' => $categories]);

}

Trier par order croissant / décroissant

Dans le controller

**public function** indexAction(Request $request)

{

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

// pour récupérer l'info qu'on a dans l'URL (ici sort, ASC)

$sort = $request->query->get("sort", "ASC");

// test au cas où l'on n'a pas ASC ou DESC pour sort

**if** (!in\_array($sort, ['ASC', 'DESC'])) $sort = 'ASC';

$brands = $em->getRepository('AdminBundle:Brand')->findBy([], ["title"=>$sort]);

**return** $this->render('brand/index.html.twig', **array**(

'brands' => $brands,

));

}

Dans le twig

<a href="{{ path("brand\_index", { sort:"ASC"}) }}">Tri A > Z</a>

<br>

<a href="{{ path("brand\_index", { sort:"DESC"}) }}">Tri Z > A</a>

Installer un bundle de fixture

* Documentation : <http://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html>
* Entrer : php composer.phar require --dev doctrine/doctrine-fixtures-bundle
* Aller dans composer.json pour vérifier (ligne 29 pour fixture)
* puis dans AppKernel.php et ajouter dans **public function** registerBundles dans le if, ligne 27 $bundles[] = new **Doctrine**\Bundle\FixturesBundle\DoctrineFixturesBundle();
* pour executer dans un fichier précis :php bin/console doctrine:fixtures:load --append –fixtures=src/AdminBundle/DataFixtures/ORM/LoadCategoryData.php
* pour tout générer : php bin/console doctrine:fixtures:load –append

Dans le fichier src/AdminBundle/DataFixtures/ORM/LoadCategoryData.php (qui est a créé) mettre le code suivant :

**<?php**

**namespace** AppBundle\DataFixtures\ORM;

**use** Doctrine\Common\DataFixtures\FixtureInterface;

**use** Doctrine\Common\Persistence\ObjectManager;

**use** AdminBundle\Entity\Product;

**class** LoadProductData **implements** FixtureInterface

{

**public function** load(ObjectManager $manager)

{

**for**($i = 0; $i < 5; $i++)

{

$product = **new** Product();

$product->setTitle('un nouveau titre')

->setDescription('lorem ipsum')

->setPrice(rand(1,100))

->setQuantity(2);

$manager->persist($product);

$manager->flush();

// php bin/console doctrine:fixtures:load (Supprimer entièrement les informations de la BDD)

// php bin/console doctrine:fixtures:load --append (Supprimer entièrement les informations de la BDD)

}

}

}

Base de donnée

Schéma et jointure

Dans le .php dans Entity

*/\*\**

*\** ***@ORM\ManyToOne****(targetEntity="Brand")*

*\** ***@ORM\JoinColumn****(name="id\_brand", referencedColumnName="id", nullable=false)*

*\*/*

**private** $marque;

* puis générer les Get / set avec : php bin/console doctrine:generate:entities AdminBundle:Product
* et mettre à jour la BDD avec **php bin/console doctrine:schema:update –forcegit push** . Il faudra faire attention aux données déjà présentes

Dans l'affichage d'un SELECT (cf fichier AdminBundle/Form/ProductTYpe.php.php au niveau de la fonction buildForm() )

* Multiple : il y a plusieur choix
* Expanded : affichage de plusieurs champs
* Bidrectionnel : qui va dans les deux sens, qui permet de faire une table de jointure
* Unidirectionnel : dans une seule direction

Jointure de plusieurs tables

* Dans l'exemple ci dessous : multidirectionnel, @ORM\, fait référence au use du haut, targetEntity, c'est sur quoi on se base, INVERSEDBY : ce qui est en retour ; @jointable : qui sur quoi on fait la jointure ; products\_categories : ce qui va etre créé en BDD

*/\*\**

*\* Many Users have Many Groups.*

*\** ***@ORM\ManyToMany****(targetEntity="Category", inversedBy="Products")*

*\** ***@ORM\JoinTable****(name="products\_categories")*

*\*/*

**private** $categories;

et de l'autre coté, sur la table sur laquelle on doit faire la jointure :

*/\*\**

*\* Many Products have Many Categories*

*\** ***@ORM\ManyToMany****(targetEntity="Product", mappedBy="Categories")*

*\*/*

**private** $products;

1. Puis mettre à jours les entités (Get / Set) avec php bin/console doctrine:generate:entities AdminBundle
2. Puis on met à jour la BDD avec

Consulter la doc : http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html

**Cascade**

Dans l'entité

*/\*\**

*\** ***@ORM\ManyToOne****(targetEntity="Brand", cascade={"remove}*

*\** ***@ORM\JoinColumn****(name="id\_brand", referencedColumnName="id", nullable=false, onDelete="CASCADE")*

*\*/*

**private** $marque;

Le premier ORM est pour Doctrine

Le Deuxieme ORM est pour PhpmyAdmin

Puis on met à jour les Get / Set et la BDD :

1. php bin/console doctrine:generate:entities AdminBundle:Product
2. php bin/console doctrine:schema:update –force

Ajouter une image en BDD

* Dans Entity/Product, rajouter :

*/\*\**

*\** ***@var*** *string*

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

*\*/*

**private** $image;

* puis générer Get/Set et mettre à jour la BDD avec
  + php bin/console doctrine:generate:entities AdminBundle:Product

et

* + php bin/console doctrine:generate:entities AdminBundle:Product
* puis dans le Form/ProductType, rajouter :

->add('image', FileType::*class*)

puis dans le ProductController :

// Transfert d'image

$image = $product->getImage();

$image->move('upload/', 'fichier.' . $image->guessExtension());

Doctrine

Pour accéder à doctrine : getDoctrine(), puis

* Repository (en utilisant getRepository() )
  + SELECT : permet de stocker ses propres requetes personnalisées. On utilisera plus createQueryBuidler() pour faire la requete
  + 4 méthodes déjà existantes : Find, findAll, findBy, FindOneBy
* Manager : pour gérer les entités (tout sauf SELECT), en utilsant getManager(). On utilisera plus createQuery() pour faire la requete
  + UPDATE
  + INSERT
  + DELETE
* flush() : exécute la requête
* persist() : met en file d'attente les différentes requêtes et c'est le flush() qui va les executer
* $em : pour entity manager, en lien avec le dossier Entity, qui permet de gérer la BDD
* getQuery():toujours en avant dernière méthodes, permet d'executer la requete
* getResult : méthode de récupération en dernier dans la requete. Celle la permet de récupèrer un tableau d'objet. Il y en a d'autre, dans la doc de doctrine
* getOnorNullResult : permet de savoir si l'élément existe ou pas
* $rc = recherche la classe de dépôt (aka repository class)
* dans le select() : le product se réfère à l'alias du createQueryBuilder, On peut mettre tous les champs, séparé par une virgule. Par exemple

->select('product.description, product.price')

* ->setFirstResult(2) : correspond au OFFSET, à partir d'où on commence la requete
* ORM : object relationner mapper
* ->createQuery : on utilise 99 % de slq, donc on écrit comme pour faire des requetes classiques en SQL, sauf pour les allias

Pour les conditions :

* ->where() : fonctionne comme le WHERE, on indique dedans l'allias et la colonne de la table qui vaut une varibal qui devra être défini dans setParameters() (on met un -s quand il s'agit d'un tableau). Et dans ce setParameters, on affiche ce qui est en variable
* ->andwhere : ajouter pour pouvoir créer de nouvelle conditions

->where('product.description = :desc')

->andWhere('product.price > :price')

->setParameters([

'desc' => 'lorem ipsum',

'price' => '298'

])

* ->join('product.marque', 'brand') : JOIN permet de faire le lien. PRODUCT, c'est l'alias à quoi on fait la jointure. MARQUE c'est la propriété de l'objet qui est dans l'entity, et le BRAND c'est l'alias de la table de l'autre partie de la jointure. Une fois fait, il faut rajouter l'élément qui fait la kjointure dans le select

Cycles de vie des entités

* Permet de récupérer des évenements selon le cycle de vie de l'entité
* Au moment où l'on fait un INSERT, alors se déclencheront :
  + prePersist :On peut récupérer l’événement juste AVANT que l'évenement soit créé dans la base.
  + postPersist : On peut récupérer l’événement juste APRES que l'évenement soit créé dans la base.
* Au moment où l'on fait un UPDATE, alors se déclencheront ::
  + preUpdate
  + postUpdate
* Au moment où l'on fait un DELETE, alors se déclencheront ::
  + preRemove
  + postRemove
* Au moment où une entité est instanciée :
  + postLoad
  + Il n'y a pas de preLoad
* Il y a d'autre type d'écouteur : preFlush
* Ecouteur : ils sont dans le fichier Listener. Le dossier listener doit écouter un type de chose, par excemple ProductListener. Le namespace correspond au nom du Bundle et \Listener.
* Créer la public function comme ça :

**public function** postPersist(Product $entity, LifecycleEventArgs $args)

{

}

* puis, dans l'entité Product, mettre dans l'annotation, tout en haut, rajouter la ligne :

*\** ***@ORM\EntityListeners****({"Admin/bundle\Listener\ProductListener"})*

* dans la fonction, en faisant '$entity->' on va récupèrer tout les évenements de l'objet '$entity'

Services

* Créer un dossier Service dans le bundle, puis en générals Utils
* Avec la commande php bin/console debug:container , cela permet d'afficher tous les services
* Pour enregistrer un service, aller dans app/confi/services/yml. Mettre 4 espaces pour la tabulation
  + Pour le nom : bundle\_dossier\_class, par exemple : **admin.service.utils.strings:**
  + pui rentrer la class : **class:** AdminBundle\\Service\Utils\StringService
* Ensuite, remplir un fonction, par exemple

**class** StringService

{

**public function** generateUniqId()

{

$result = bin2hex(openssl\_random\_pseudo\_bytes(16));

**return** $result;

}

}

* Et enfin dans le controller, appeler le service en faisant :

// service utils

$serviceUtils = $this->get('admin.service.utils.string');

* Dans confi.yml : on stoke les paramètres de l'application

Fonctions Twig

* Créer un Fichier Twig à la racin du bundle
* Créer une class, en mettant bien le namespace
* faire un extends de la class Twig\_Extension : **class** AppExtension **extends** \Twig\_Extension
* Dans la fonction, faire un clic droit > Generate > Overide Methods > Twig > GetFunction
* Il faut ensuite redéfinir en rajoutant dans le return le news \Twig\_SimpleFunctions(), pour obtenir ça :

**namespace** AdminBundle\Twig;

**class** AppExtension **extends** \Twig\_Extension

{

**public function** getFunctions()

{

**return** [

**new** \Twig\_SimpleFunction();

]

}

}

* Dans SimpleFunction(), le premier paramètre dans les () est pour les vues qui vont être utilisées dans la vue twig ; le 2eme est un callable , comme dans AddEventListener(« click », fonction). Il n'y a donc pas de () a la fonction, car elle est excécutée au moment de l'exécution du 1er paramètre
* Ensuite dans service.yml , rentrer un service, cf :

**admin.twig.extension:**

**class:** AdminBundle\Twig\AppExtension

**arguments:** [ ]

**tags:**

- { **name:** twig.extension }

* Puis après on peut l'afficher dans une vue de twig

Gestion des utilisateurs et sécurité

Doc : [http://symfony.com/doc/current/security.html#loading-users-from-the-database](http://symfony.com/doc/current/security.html" \l "loading-users-from-the-database) et <http://symfony.com/doc/current/security/entity_provider.html>, prendre le code de Create your User Entity, puis dans AppBundle/Entity, créé un fichier PHP class, et coller tout (sauf laisser le <?php)

Changer :

*\** ***@ORM\Table****(name="app\_users")*

par

*\** ***@ORM\Table****(name="app\_user")*

Puis créer ou mettre un dossier Repository et créer un fichier UserRepository et mettre à jour en fonction.

Dans le fichier security.yml, rentrer les éléments qui sont dans le code dispo ici Configure Security to load from your Entity

Entités User

Fichier security.yml, 3 é&léments :

1. Les enités
2. Les roles
3. Les routes protégées
4. formulaire de connexion

* Role : niveau d'accès.
* Pour créer un rôle : ROLE\_USER ou ROLE\_ADMIN dans le fichier security.yml

Entités mâle

Connexion

Création de formulaire : http://symfony.com/doc/current/security/form\_login\_setup.html

Création de compte

Voir les dossier avec Security.yml, security.login.html.twig, controller User / Role,

Deconnexion

Chercher dans <http://symfony.com/doc/current/security.html> lougout, et copier ca dans security.yml

**logout:**

**path:** /logout

**target:** /

Puis, dans SecurityController, créer une fonction lougout, avec ces éléments, et cette fonction est vide

Création du formulaire d'utilisateur

* Dans le terminal, rentrer : php bin/console doctrine:generate:form AppBundle:User
* Cela va créer un formulaire dans le Bundle.
* Modifier le $builder en fonction
* Puis faire le formulaire du twig (signup)
* puis adapter la fonction isSubmitted, comme suit (ps : les traitement doivent etre fait dans les ecouteur, pas dans les controllers)

**if**($form->isSubmitted() && $form->isValid())

{

$data = $form->getData();

$encodePassword = $this->get('security.password\_encoder');

$password = $encodePassword->encodePassword($data, $data->getPassword());

$data->setpassword($password);

$role = $rcRole->findOneBy([

'name' =>'ROLE\_USER'

]);

$data->addRole($role);

// Pour préarer la requête

$em->persist($data);

// Pour envoyer la requête

$em->flush();

**return** $this->redirectToRoute('public\_homepage');

}

* Mettre les message / restriction dans le twig + dans les entités

Faire les emails d'inscription

* Il ne faut pas que l'utilisateur soit inscrit directement, c'est pour ça qu'il faut lui envoyer un email

Traduction

* Locale : /fr ou /en , ce qu'il y a dans l'url. Dans le fichier config.yml, c'est en par défaut
* I18n : internationalisation
* Il faut internationaliser, en formattant.
  + Date
  + Monnaie
  + Numbers
* Il y a des extensions twig qu'il faut installer : <https://packagist.org/packages/twig/extensions>
* rentrer la ligne de commande dans packagist
* installer ensuite Intl
* Aller dans services.yml et intégrer les lignes suivantes :

**twig.extension.intl:**

**class:** Twig\_Extensions\_Extension\_Intl

**tags:**

- { **name:** twig.extension }

et après

**translator:** { **fallbacks:** ["%locale%"] }

Puis dans les vues, mettre des {{ 1000 | localizednumber }} .

Il faut ensuite préfixe les routes, va le fichier routing.yml, ic

**admin:**

**resource:** "@AdminBundle/Controller/"

**type:** annotation

**prefix:** /{\_locale}

Il faut utiliser le code langue et non le code pays

Pour metre les fichiers de traduction, le smettre dans App/Ressources/translations, créé les fichiers de traduction la avec message.en.yml

Ensuite, remplirces fichiers avec les clés (aqui sont les memes dans le fichiers, un peu comme de svariables), et la valeur qui change en fonction du fichier de traductiobn

Dans Twig, pour intégrer ce texte traduit, il faut insérer {{ 'clé' | trans }}

On rajoute un nouveai fichier de traduction, il faut vider son cache

Pour créer des variabale, dans le twig mettre

<p>{{ 'text.placeholder' | trans( { '%name%' : 'Alexis' }) }}</p>

* messages.fr.yml =
  + message= domaine
  + .fr= locale
  + .yml = format

Sélecteur de langues

Rajouter dans config.yml :

**locales:** [fr, en]

et aussi :

**twig :**

**globals:**

**locales:** '%locales%'

Contraintes de validation

Elle doivent etre obligatoire dans le fichier Ressourecs/translations/validators.fr.yml ou validators.en.yml

Attribute

Voir la documentation twig : <http://twig.sensiolabs.org/doc/2.x/functions/attribute.html>

Pour récupérer la langue du navigateur :

**public function** indexActions(Request $request)

{

dump($request->getLanguages()); **exit**;

}

Gestion des requetes et des réponses

Documentation ici : <http://symfony.com/doc/current/components/http_kernel.html>

Le but des Kernel request, est d'intercepter les requetes HTTP

Cycle de vie d'une requete symfony :

| Name | KernelEvents Constant | Argument passed to the listener |
| --- | --- | --- |
| kernel.request | KernelEvents::REQUEST | [GetResponseEvent](http://api.symfony.com/3.2/Symfony/Component/HttpKernel/Event/GetResponseEvent.html) |
| kernel.controller | KernelEvents::CONTROLLER | [FilterControllerEvent](http://api.symfony.com/3.2/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html) |
| kernel.view | KernelEvents::VIEW | [GetResponseForControllerResultEvent](http://api.symfony.com/3.2/Symfony/Component/HttpKernel/Event/GetResponseForControllerResultEvent.html) |
| kernel.response | KernelEvents::RESPONSE | [FilterResponseEvent](http://api.symfony.com/3.2/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html) |
| kernel.finish\_request | KernelEvents::FINISH\_REQUEST | [FinishRequestEvent](http://api.symfony.com/3.2/Symfony/Component/HttpKernel/Event/FinishRequestEvent.html) |
| kernel.terminate | KernelEvents::TERMINATE | [PostResponseEvent](http://api.symfony.com/3.2/Symfony/Component/HttpKernel/Event/PostResponseEvent.html) |
| kernel.exception | KernelEvents::EXCEPTION | [GetResponseForExceptionEvent](http://api.symfony.com/3.2/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html) |

Pour gérer ça, il faut créer un souscripteur (qui va gérer plusieur évemement, VS écouteur qui écoute un évenement).  
Quand on créé un souscripteurs :

* Dans la classe, implémenter la bonne interface
* Dans services.yml, bien mettre le services

Quand on a bien paramètrer notre service, on peuyt l'utiliser dans le souscripteur. Par contre, pour utiliser un autre service dans ce service, il faut mettre l'autre service en arguement dans le service.yml.

Il faut ensuite créer le paramètre de classe en faisant un constructeur

Récupérer l'ip

aller sur le site : <http://www.webservicex.net/> et indiquer dans l'URL geoipservice.asmx/GetGeoIP?IPAddress=89.227.222.139, l'ip est a la fin

* php bin/console debug:event-dispatcher : liste des évenments qui sont disponibles dans symfony

Gestion du bandeau des cookies et des sessions

Dans $session = $request->getSession() ; Dans un controller)

* →get : récupérer la valeur d'une clé
* →set : créer une clé
* →clear : pour vider la session
* →has : tester une clé
* →remove : supprimer une clé

et dans twig, app.session

Renvoyer du JSON

Dans le Subscriber

*/\*\**

*\** ***@Route****("/disclaimer-cookies", name="disclaimer-cookies")*

*\*/*

**public function** disclaimerCookiesAction(Request $request)

{

$disclaimer = $request->get('disclaimer');

$session = $request->getSession();

$session->set('disclaimer', $disclaimer);

**return new** JsonResponse([

'success' => 'ok'

]);

// dump($session);exit;

}

Dans le TWIG

{% **block** javascript %}

<script>

**var** btnCookie = $('.Cookies a');

btnCookie.on('click', clicBtnCookie);

**function** clicBtnCookie(e) {

e.preventDefault();

$.ajax({

method: 'POST',

url: 'http://localhost/symfony/web/app\_dev.php/disclaimer-cookies', // méthode récupérée par la route

data: 'disclaimer=1', // Quelles sont les informations qu'on envoie

success: onSuccess

});

**function** onSuccess(data){

console.log(data);

}

}

</script>

{% **endblock** javascript %}

**public function** createCartActions(Request $request)

{

$session = $request->getSession();

**if**(!$session->has('cart'))

{

$session->set('cart', [

'id' => [],

'qty' => [],

]);

}

$this->render('Public/Products/show.html.twig');

}

Evenement personalisé

Pour déclencher des ations en fonction d'une volonté précise

1. Créer un dossier Event

2. Créer un Listeur d'évenement : VisittEvents.php

3. Créer un fichier lié à un évenement : VisitContactEvent.php

4. Créer le souscripteur, la classe qui écoute les évenmeent multiple

5. Enregistrer le souscripteur dans le service.yml et vérifier dans la console avec la commande : php bin/console debug:event-dispatcher

6. On déclenche l'évenement dans le CONTROLLER avec la fonction dispatch()

Créer ses propres commandes

Permet d'accéder au container de Symfony (ajouter un rôle à l'utilistaeur, exporter la BDD, envoyer un mail à des perosnnes qui sont pas connecté depuis 3 mois…) → Faire des actions, sans passer par une interface visuel

1. Créer un dossier Command dans un bundle,
2. Créer une classe par commande dans ce dossier, par ex : UserAddRoleCommand.php
3. Faire un extends du ContainerAwaraCommand
4. Générer des methods to override, aller dans commmand et sélectionner configure et execute
   1. Configure : Configure la commande, avec le
      1. Nom,
      2. description,
      3. arguments (ce qui suit la commande, par exemple doctrine:generate:entites AppBundle → AppBundle est l'argument)
      4. options : --force par exemple, qui a un statut REQUIRED
   2. Execute : On récupère les argument du Configure
      1. Input :
      2. Output :
5. Pour voir les ligne de commandes, faire dans le terminal : php bin/console, et vérifier
6. Pour changer les rôle ou executer faire php bin/console app:user:role:add Alexis ADMIN

Tester le site

Télécharger <https://phpunit.de/>

Ensuite il faut copier les dossiers / fichiers qu'on veut tester

Il faut préfixer les classes par test

et faire en ligne de commande : phpunit

* Test unitaire : On teste une seule classe → un services
* Test fonctionnel : on teste une fonctionnalité

Pour tester des fonctions, il faut utiliser assert...(), qui contient plusieurs paramètre :

1. Résultat attendu
2. Le traitement

* $client : La variable simule le navigateur utilisé par une vraie personne
* $crawler :