1) INTRODUCTION

Symfony 2, en 2011, est le premier framework un peu moderne de PHP. Il a une approche façon boîte à outils, un peu moins monolithique que ses concurrents. Il est composé de différents composants (comme Twig, le router, les formulaires etc) qui liées ensemble, permettent de créer une applications web de manière plus rapide et de manière « standardisée » en forçant les bonnes pratiques aux maximum.

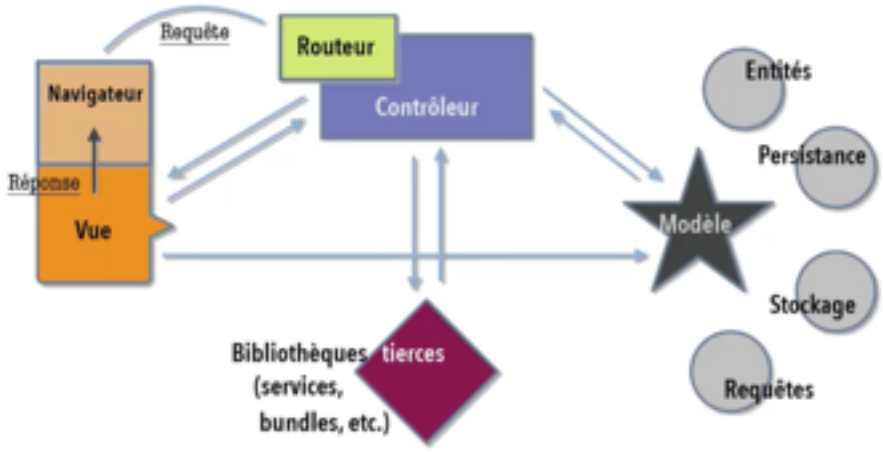
La version 4 pousse encore plus cette approche de boite à outils avec une installation de départ d'un squelette Symfony, presque vide de tout composant, que l'on peut ensuite ajouter au fur et à mesure si besoin.

Aujourd'hui de nombreuses applications utilisent des parties de cette boite à outils : Drupal, Laravel, Prestashop, Magento, PHPBB.

1) Piqure de rappel (en bref) :

- PHP 5.3 a introduit les espaces de nom, alias Namespace, pour éviter les collusions de noms. On peut aussi définir des alias / raccourcis de classe pour améliorer la lisibilité. On le définie dans la première ligne PHP et on respecte l'arborescence des dossiers. L'autoload permet de venir charger automatiquement ces classes.
- PSR : C'est une initiative pour normaliser le code PHP avec des recommandations, des conventions, mis en place par le FIG (Framework Interoperability Group).
  La plus importante est le PSR-4, elle précise comment se fait l'autoloading des classes. Mais il y a de nombreuses recommandations.
- MVC : architecture de code permettant de séparer et organiser les fichiers d'une applications en fonction de leur rôle.
  - Contrôleur : reçoit les requêtes. Chargé de gérer les interactions entre l'utilisateur et le modèle. C'est le chef d'orchestre de l'application.
  - Modele : gère la manipulation et le traitement des données.
  - Vue : présente les données du modèle à l'interface utilisateur. Enregistre les actions utilisateurs (envoi d'un formulaire etc, et les transmet au contrôleur.

2) **Cycle de vie d'une requête HTPP dans une application Symfony :**

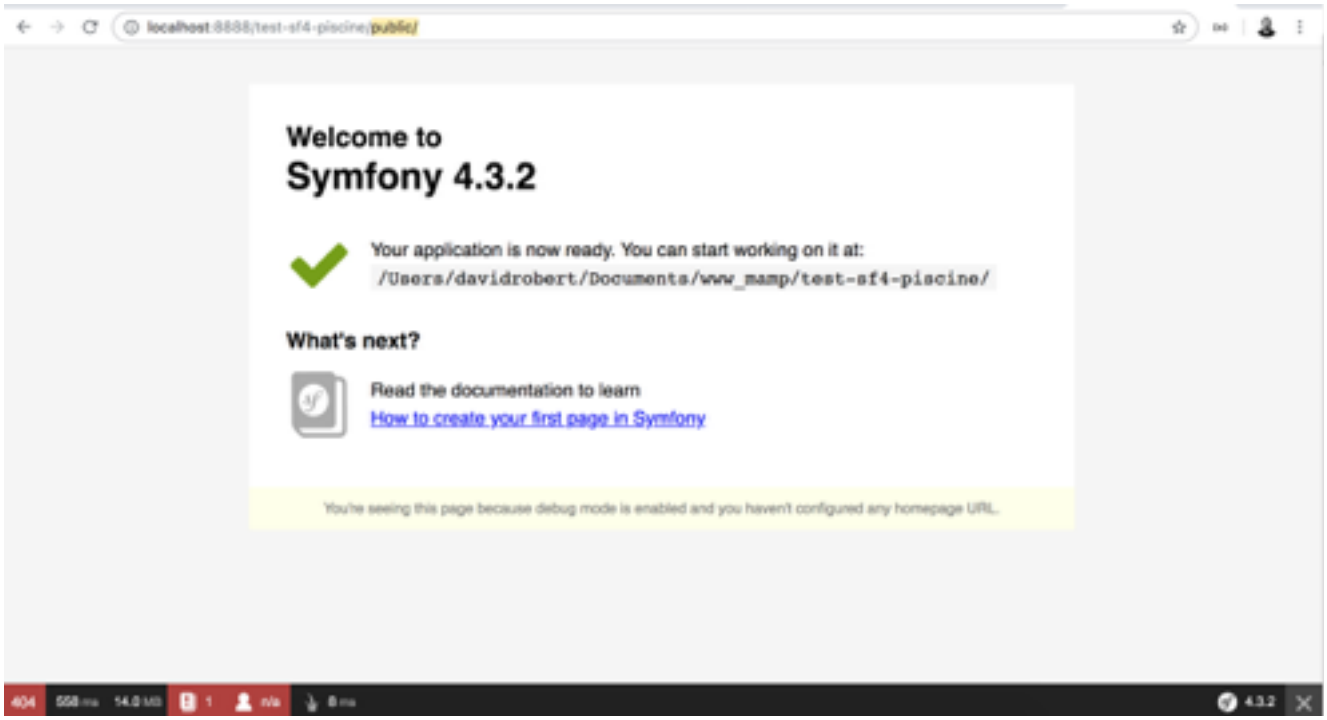


3) Installation de Symfony

En ligne de commande :
composer create-project symfony/website-skeleton nomduprojet

Si Apache est utilisé (notamment avec MAMP), il faut configurer le .htaccess Pour ça on va utiliser :
composer require symfony/apache-pack

Une fois installé, un message de bienvenu devrait s'afficher en allant sur votre projet via votre serveur local, suivi de « public » :



3) Architecture

- Le dossier « public » est le seul dossier accessible par le navigateur. Il contient un fichier index.php qui est le point d'entrée de l'application.

- Le dossier config, contient les fichiers de configuration de chacun des packages de l'application.
- Le dossier src contient votre code
- Le dossier templates contient les fichiers Twig (la partie « vue » de votre application)
- Le dossier var contient les logs et les caches
- Le dossier tests contient les tests écrits pour votre application
- Le dossier bin contient le fichier d'entrée pour la ligne de commande
- Le dossier vendor contient les bibliothèques
- Le dossier translations contient les traductions de vos pages (si votre applications est en plusieurs langues)
- Le fichier .env contient la configuration pour l'accès à la base de données et d'autres informations sens

Environnements disponibles dans Symfony :

- prod : le site visible par les utilisateurs. N'affiche pas les erreurs et utilise des fichiers en cache pour optimiser les performances.
- dev : pour développer. Permet notamment d'afficher les erreurs et la barre de débug.
- test : utilisé pour les jeux de tests.

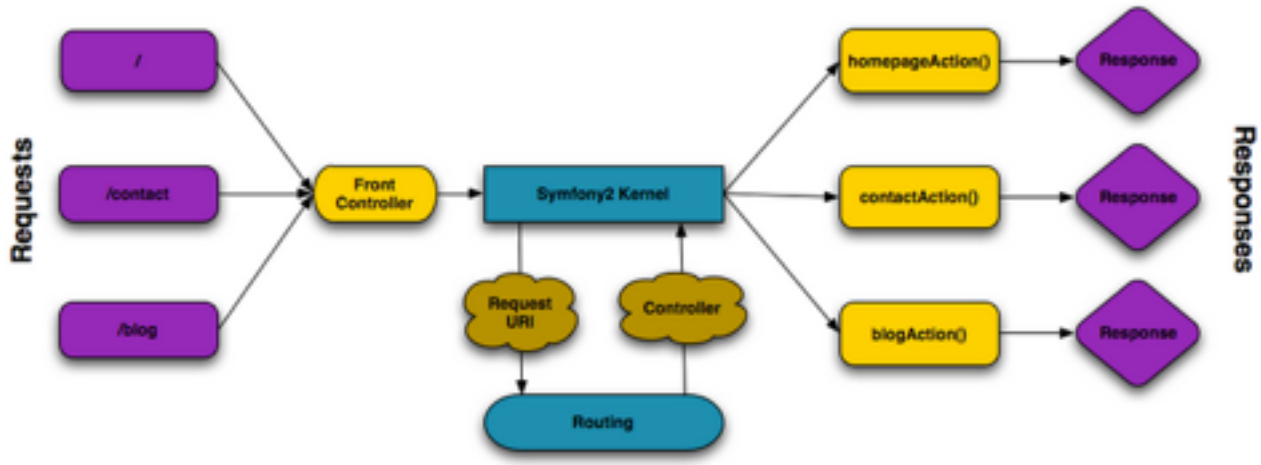
Chaque environnement peut avoir une configuration différente (via des fichiers de configurations spécifiques dans le dossier config).

2) LES ROUTES

Les routes dans Symfony permettent de définir une URL, d'identifier une ressource.

Une URL se compose de plusieurs éléments :

- protocole : http
- nom de domaine : www.example.com
- un port : le serveur écoute sur un port
- un chemin : spécifie le contenu auquel on accède via une arborescence
- un nom de ressource : index.html, image.jpg etc
- des options : ?nom=david&prenom=robert



Le composant routing dans symfony :
- gère les collections de routes
- détermine les contextes de requêtes (analyse l'url pour savoir le chemin demandé etc)
- relie une url avec une route

Les routes sont un peu l'API de notre appli. A chaque route fait référence à une méthode de controleur.

Une route contient deux propriétés principales :
- id : un identifiant unique qui permet à l'appli d'accéder à la route.
- path : le schéma de l'url qui permettra d'analyser la requête du navigateur.

Les routes s'écrivent en « annotations » au dessus des méthodes de contrôleurs (il est possible de les écrire aussi dans un fichier Yaml, XML ou même PHP). Une annotation est un commentaire PHP qui permet d'exécuter du code.

Pour installer le composant Annotations :
composer require annotations

TP : créer deux routes, qui viennent afficher deux var_dump différents en utilisant ce squelette de controleur :

```
<?php

namespace App\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HomeController extends Controller

{

        public function index()

        {

                dump('hello'); die;

        }

}
```

3) CONTROLEURS

Une classe controleur dans Symfony doit hériter de la classe Symfony\Bundle \FrameworkBundle\Controller\AbstractController.
Le rôle du contrôleur est de recueillir une requête http (filtrée par le routeur), d'effectuer un traitement (récupérations de données dans la bdd, etc) puis de rendre du contenu via la view.

Chaque controleur doit retourner une réponse compatible avec le format http. Pour cela; il faut retourner une instance de la classe Response du composant HttpFoundation.

Objet Request

La classe request = permet de gérer tous les paramètres gérés habituellement par PHP dans les variables super-globales ($_GET, $_POST etc).

Pour l'utiliser, on passe l'objet en argument de l'action controleur (avec un type hinting vers l'interface ou la classe de Request, qui nous permet d'éviter d'avoir à instancier l'objet Request à la main).

Et on peut venir récupérer, avec les getters de la classe Request, tout ce qu'on veut lié à la requête : getHost, getMethod, getClientIp, getScheme (http, https etc), la langue, le chemin etc.

TP : créer une nouvelle route, et dans un controleur récupérer et afficher l'url + le protocole utilisé pour la requete.

Pour récupérer des parametres GET, on utilise $request->query->get('nomdu parametre');

TP : créer une nouvelle route avec des paramètres GET, un chiffre et un bool, et dans un controleur faire une condition en fonction du bool pour afficher le nombre.

Quand on définit une route, on peut ajouter dans le chemin choisi des variables (nommées wildcards) :

Exemple :

```
/**
 * @Route(«/blog/{id} », name=« blog_article »)
 */
```

Exemple d'url :

/articles/1
/articles/2

Pour récupérer ces « wildcards », il suffit de passer en argument du controleur une variable du même nom que la wildcard, donc ici « $id »

On pourrait les rendre optionnelles via une entrée default dans nos routes. On pourrait aussi mettre des contraintes sur ces variables pour qu'elles respectent un certain schéma, via une entrée « requirement » dans nos routes.

TP : créer une route avec des variables $categorie et $titre, et les afficher dans du html en les ajoutant via le controleur avec une response.

TP : créer une route avec des variables âge, $nom $prenom et faire une condiition si l'age est inférieur à 18 ans afficher un message de refus, sinon afficher le nom et prénom.

Réponses et redirections

Types de réponses qu'un controleur peut renvoyer :
- render : renvoie vers une view
- redirect : renvoie vers un autre controleur
- redirectToRoute : renvoie vers une route
- forward

TPC : créer une route, avec un render de view

TP : créer une nouvelle route, avec un render de view très simple

On peut créer une url dans un controleur avec $this->generateUrl avec en paramètre le name d'une route.
Pour créer une redirection, on peut utiliser la méthode $this->redirect, avec en parametre l'url vers laquelle on veut rediriger.

TP : créer une nouvelle route, et un controleur et dans le controleur faire un redirect vers l'url de la home page
TP2 : créer une nouvelle route avec un parametre et un controleur et dans le controleur, si le parametre existe rediriger vers la home, sinon rendre une view.

On peut aussi rediriger directement vers une route avec la méthode $this->redirectToRoute().

Commandes

Symfony fourni des commandes qu'on peut utiliser pour éviter certaines tâches répétitives (créations de fichiers, etc) ou débuguer.

On peut voir la liste de toutes nos routes avec bin/console router:debug
On peut aussi tester que toutes les routes sont accessibles sans doublons via bin/console router:match suivi du chemin qu'on veut tester/

Bonnes pratiques :

Créer une classe de controleur par fonctionnalités.
Organiser des groupes de controleurs par dossier.
Ne pas utiliser les controleurs pour abriter du code lié à la présentation ou à la base de données.

TWIG

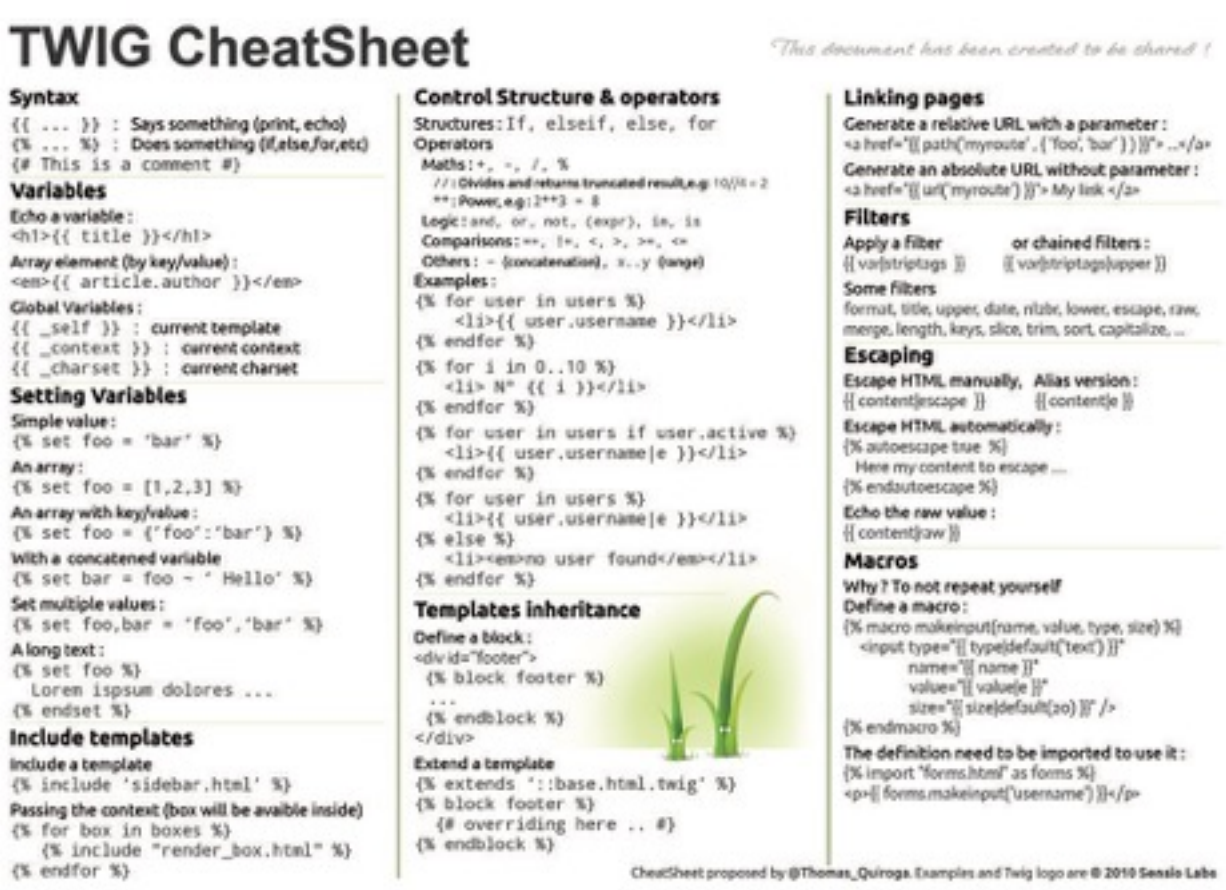
Présentation :

- Pseudo langage PHP plus accessible aux non-développeurs
- Permet de créer des hiérarchies de templates (des fichiers « base.html.twig » qui va pouvoir être utilisé comme base dans tous les autres templates, avec des « blocks » qu'on peut remplir en fonction du template.

La balise d'ouverture PHP « <?php » et la balise de fermeture « ?> » sont remplacées par « {{« et « }} »

Syntaxe :

Dans twig, les objets et les tableaux sont utilisés de la même manière.
- on peut utiliser des conditions, des boucles etc d'une manière similaire au PHP
- commentaire
- filtre et fonction, permettent de modifier la valeur ou présentation de certaines variables
- extends : permet de faire de l'héritage de template



TWIG CheatSheet

"This document has been created to be shared !"

Syntax

{{ ... }} : Says something (print, echo)
{% ... %} : Does something (if,else,for,etc)
{# This is a comment #}

Variables

Echo a variable :
<h1>{{ title }}</h1>

Array element (by key/value) :
{{ article.author }}

Global Variables :
{{ _self }} : current template
{{ _context }} : current context
{{ _charset }} : current charset

Setting Variables

Simple value :
{% set foo = 'bar' %}

An array :
{% set foo = [1,2,3] %}

An array with key/value :
{% set foo = {'foo':'bar'} %}

With a concatened variable
{% set bar = foo ~ ' Hello' %}

Set multiple values :
{% set foo,bar = 'foo','bar' %}

A long text :
{% set foo %}
  Lorem ispsum dolores ...
{% endset %}

Include templates

Include a template
{% include 'sidebar.html' %}

Passing the context (box will be avaible inside)
{% for box in boxes %}
  {% include "render_box.html" %}
{% endfor %}

Control Structure & operators

Structures : If, elseif, else, for

Operators

Maths : +, -, /, %
//: Divides and returns truncated result,e.g: 10//4 = 2
: Power, e.g: 23 = 8

Logic : and, or, not, (expr), in, is

Comparisons : ==, !=, <, >, >=, <=

Others : ~ (concatenation), x..y (range)

Examples :
{% for user in users %}
  {{ user.username }}
{% endfor %}

{% for i in 0..10 %}
   N° {{ i }}
{% endfor %}

{% for user in users if user.active %}
  {{ user.username|e }}
{% endfor %}

{% for user in users %}
  {{ user.username|e }}
{% else %}
  no user found
{% endfor %}

Templates inheritance

Define a block :

 {% block footer %}
 ...
 {% endblock %}

Extend a template
{% extends '::base.html.twig' %}
{% block footer %}
  {# overriding here .. #}
{% endblock %}

Linking pages

Generate a relative URL with a parameter :
 ...

Generate an absolute URL without parameter :
 My link

Filters

Apply a filter
{{ var|striptags }}

or chained filters :
{{ var|striptags|upper }}

Some filters
format, title, upper, date, nl2br, lower, escape, raw, merge, length, keys, slice, trim, sort, capitalize, ...

Escaping

Escape HTML manually,
{{ content|escape }}

Alias version :
{{ content|e }}

Escape HTML automatically :
{% autoescape true %}
  Here my content to escape ...
{% endautoescape %}

Echo the raw value :
{{ content|raw }}

Macros

Why ? To not repeat yourself
Define a macro :
{% macro makeinput(name, value, type, size) %}
  <input type="{{ type|default('text') }}"
      name="{{ name }}"
      value="{{ value|e }}"
      size="{{ size|default(20) }}" />
{% endmacro %}

The definition need to be imported to use it :
{% import "forms.html" as forms %}
{{ forms.makeinput('username') }}

CheatSheet proposed by @Thomas_Quiroga. Examples and Twig logo are © 2010 Sensio Labs

TP : exercice variable : faire passer une variable depuis le controleur et l'afficher dans le twig

TP : exercice if : faire passer une variable qui contient un booleen depuis le controleur et l'afficher dans le twig

TP : exercice for : faire passer une variable qui contient un array depuis le controleur et l'afficher dans le twig

TP : ajouter dans l'url une variable booleen, la récupérer avec le controleur et s'en servir avec un if dans la view

On peut utiliser dans twig des filtres sur les variables qui vont permettre d'en modifier le contenu. Par exemple le filtre « upper » : {{ 'welcome' | upper }}

Liens

Pour créer des liens, on peut utiliser la fonction path avec en parametre le nom d'une route.

Ressources

On inclut des assets via la fonction asset.

On vient inclure nos assets dans le dossier public, puis par exemple css/style.css et après on les charge avec {{ asset('css/style.css') }}

Include

On peut faire des includes vers des fichiers partiels twig, grâce à la fonction include.

TP : faire un include vers un autre bout de code twig

Lien avec parametre

La fonction path prend un second argument qui est un objet et qui permet de les valeurs des wildcards à inclure dans l'url.

TPC : faire un lien depuis la page de liste des articles vers un article en particulier

TP : Faire un blog complet page d'accueil, page de liste des article, page article en utilisant les includes, les paths, link, asset etc.

Héritage

On peut créer des hiérarchies de templates (des fichiers « base.html.twig ». Le fichier base.html.twig va pouvoir être utilisé comme base dans tous les autres templates, avec des « blocks » qu'on peut remplir en fonction du template.

TP : Création d'un squelette pour voir l'héritage.

TP 2 : Création d'une base avec sidebar

TP : exo complet avec front de 3/4 pages avec https://gist.githubusercontent.com/planetoftheweb/98f35786733c8cccf81e/raw/f3dad774ed1fe20b36011b1261bb392ee759b867/data.json

LES MODELES

Beaucoup de langages de programmations incluent le paradigme de la POO. Mais pas en BDD, le modèle dominant reste relationnel.

Pour faire en sorte qu'une application construite en langage objet soit facilement intégrable avec des infos de la base de données, en relationnelles, on utilise Doctrine.

Doctrine est un ORM (object relational mapping).

Un ORM permet :
- de transformer un objet en un ensemble de relation pour pouvoir les utiliser en SQL, et inversement.
- d'assurer la persistance des données, à savoir la cohérence entre l'état de l'application et celui de la base de données.

Désavantages :
- l'application est plus lente à communiquer avec la base de données.

Construction du modèle

- Installer doctrine :

```
composer require symfony/orm-pack
composer require --dev symfony/maker-bundle
```

- Modifier le .env :

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
```

devient :

```
DATABASE_URL=mysql://root:root@127.0.0.1:3306/exercice_symfony
```

- Créer la base de données :

```
php bin/console doctrine:database:create
```

Entités

Elles se trouvent dans le dossier Entity dans le fichier src. Les entités sont souvent accompagnées par une classe de requête (repository) qui porte le nom de l'entité, suffixée par « Repository ».

—> Créer une entité « Article.php »
Toutes les entités doivent être déclarées à Doctrine via des annotations dans le fichier de la classe d'entité.

La clé primaire :

```
/**
 * @ORM\Id()
 * @ORM\GeneratedValue()
 * @ORM\Column(type="integer")
 */
private $id;
```

C'est la structure minimale de notre entité. Ensuite il faudra créer les autres champs avec les éléments fields, de la même structure que l'id :

```
/**
 * @ORM\Column(type="string", length=255)
 */
private $title;
```

Pour créer le schema de bdd qui lui correspond (créer les tables, etc…) :
—> clique droit dans l'entité —> « Generate » —> « Getters and setters »

On peut venir en suivant ajouter autant de nouvelles propriétés qu'on le veut dans notre déclaration d'entités, régénérer les entités et mettre à jour le schema de bdd.

Pour faire un update du schema :

```
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

Pour générer des entités de façon plus rapide, il existe une ligne de commande :

```
php bin/console make:entity
```

Cette commande nous permet de créer des entités de manière interactive.

```
Class name of the entity to create or update (e.g. VictoriousChef):
> Article

Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):
> title

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Article.php

Add another property? Enter the property name (or press <return> to stop adding fields):
> content

Field type (enter ? to see all types) [string]:
> text

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Article.php
```

Le champs id se crée automatiquement, nous n'avons pas besoin de le renseigner. Une fois cette étape terminée, une nouvelle entité avec ses champs va se générer (le repository va aussi se créer)

Les langages de requêtes

Quand on créé une entité, on a des classes repository qui sont également créées. C'est dans ces classes que vont se trouver toutes les requêtes pour extraire les données (ajout etc on le fait directement via le controleur). Pour les requêtes simples on peut utiliser les méthodes fournies par les repo, pour des requêtes un peu plus complètes, on doit les créer avec une syntaxe particulière.

Pour accéder aux classes de requêtes depuis le controleur, on doit instancier l'entity manager (ou mieux, utiliser le système d'Autowire de Symfony), et depuis cette classe on peut récupérer le repo avec la méthode getRepository().

Récupérer des éléments depuis la bdd

On utilise le repository qui possède déjà des méthodes implémentées de base (grâce à l'héritage).

Exemple :

```
$repository = $this->getDoctrine()->getRepository(Auteur::class);

$auteurs = $repository->find($auteurId);

$auteurs = $repository->findAll();
```

Query builder

Le query Builder Permet de créer des requêtes SQL (dans la classe de requête) en utilisant le langage objet. On l'initialise avec l'instruction $qb = $this->createQueryBuilder('u');

Dans le query builder on peut utiliser des méthodes sur l'objet $qb : select, update, join, where andWhere, orderBy etc.
Il n'y a pas de clause from, car le parametre 'u' (qui peut être modifié) fait référence à l'entité liée au repository.

TP : créer une méthode dans le repo pour récupérer les livres liés à un genre.

```
public function getByGenre($genre)
{
    $qb = $this->createQueryBuilder('u');
    $query = $qb->select('u')
                ->where('u.genre = :genre')
                ->setParameter('genre', $genre)
                ->getQuery();
    $resultats = $query->getArrayResult();
    return $resultats;
}
```

Voir l'équivalent avec les méthodes findBy.

TP : créer une méthode dans le repo pour trouver par auteur et l'afficher dans une view nouvelle avec une route nouvelle. Faire les deux solutions : aussi implémenter avec findBy.

TPC : créer une méthode dans le repo pour trouver les livres par genre par la première lettre

```
public function getByGenreFirstLetter($letter)
{
    $qb = $this->createQueryBuilder('u');
    $query = $qb->select('u')
                ->where('SUBSTRING(u.genre, 1, 1) = :letter')
                ->setParameter('letter', $letter)
                ->getQuery();
```

```
        $resultats = $query->getArrayResult();

        return $resultats;

    }
```

TP : créer une méthode pour trouver les livres par auteur et via les deux premières lettres.

TPC : créer une méthode pour trouver les auteurs par un mot dans le nom :

```
        $query = $qb->select('u')
                    ->where('u.biography LIKE :word')
                    ->setParameter('word', '%'.$word.'%')
                    ->getQuery();
```

TP : Faire pareil pour trouver un mot dans la bio.

DQL

On peut aussi créer des requêtes en utilisant DQL, un langage très proche du sql.
Doctrine travaille en objet avec ce code DQL et le transforme après en SQL et l'optimise.

Dans la classe de requete on fait appel à l'entity manager, pour on utilise la méthode createQuery. Et on fait à l'intérieur comme du sql normal, sauf qu'on fait non pas référence à des tables et des colonnes, mais à des objets et propriétés d'objet.
DQL supporte aussi les parametres.
Le DQL est plus modulable sur les valeurs de retour que le query builder.

Native Queries

Les natives queries nous permettent de nous extraire du query builder ou du DQL pour écrire du SQL plus classique. Dans ces cas là, on utilise Doctrine uniquement dans le but de mapper les résultats de la requête sur les propriétés des objets / entités.

Pour utiliser les natives queries, on utilise l'entity manager avec la méthode createNativeQuery() et en premier parametre on passe du SQL.

A la fin il faut réinitialiser les requêtes avec $em->clear();

Les natives queries permettent de créer des requêtes potentiellement plus complexes que qu'avec les autres solutions de Doctrine. L'execution est également plus rapide, mais c'est par contre plus lourd à mettre en place.

Persistance

Doctrine sépare les requêtes vers la base de données en deux parties :
- le repository pour récupérer les données
- la persistance pour insérer et maj

Les outils qui gèrent ça s'appellent des moteurs de persistance. La Persistance est la sauvegarde de l'état des données d'une appli, mais aussi la restauration de leur état en cas d'erreur.
On a pour les ORM un travail également de modification des données car on passe d'objet à relationnel. Cela se fait en deux étapes, et le pilier de ce travail est l'unité de travail.

L'unité de travail récupère les objets créés, modifiés ou détruits par le controleur. et envoie des requêtes (transactions) à la bdd pour mettre à jour les données. En cas d'erreur, l'unité de travail est capable de restaurer l'état des données d'avant l'erreur. C'est donc l'unité de travail qui maintient la synchronisation entre l'état de l'application et celui de la bdd. Chaque entité qui doit être ajoutée ou modifiée en base de données doit être déclarée par l'unité de travail (grâce à la méthode $entityManager->persist()). Les entités sont manipulées dans l'unité de travail jusqu'au moment ou le modèle est mis à jour grâce à la méthode flush();

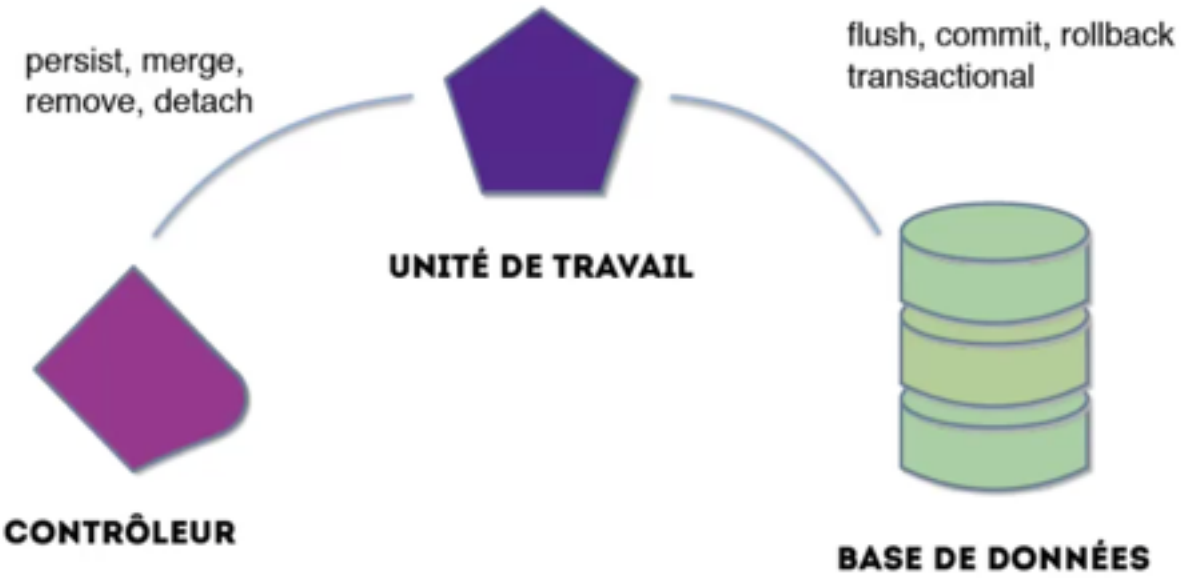
TP : créer une méthode de controleur qui va ajouter une entrée dans notre table auteur.

TP : créer une méthode qui ajoute des livres dans la table livre. ajouter 5 livres.

On peut aussi supprimer des entrée dans la bdd, en utilisant la méthode $entityManager->remove($entity);

TPC : créer une méthode qui supprime une entrée dans les auteurs

TP : créer une méthode qui supprime une entrée dans les auteurs, et définir l'entrée concerné via un segment d'url.



Pour modifier une entrée, c'est un mélange entre les deux méthodes précédentes : on récupère une entité, on utiliser les setters pour modifier ce qu'on veut. Et on re-enregistre l'entité en base de données.

TP : créer une méthode qui modifie une entrée dans les auteurs.
TP : creer une méthode qui modifie l'entrée concerné via un segment d'url.

On a également d'autres méthodes, que l'on utilise moins souvent :
- detach() qui permet de supprimer une entité de la liste des entités à synchroniser par l'unité de travail. On l'utilise avant flush().
- clear() : vide complètement l'unité de travail (ps : l'unité de travail est automatiquement

vidé quand on fait un flush()).
- merge() : permet de dire que si l'entité existe déjà mais qu'elle a été sortie de l'unité de travail, on la regroupe avec celle existante.

Les associations

- 1 - 1 (ex : un client, un panier et réciproque)
- 1 - n (ex : un article, plusieurs commentaires mais pas le contraire)
- n - n (ex : une personne va à plusieurs spectacles et un spectacle peut accueillir plusieurs personnes)

Il y a deux directions dans une association : direction dominante et direction inverse (la direction inverse est facultative, donc la relation peut être bi-directionnelle ou uni-directionnelle).

Pour créer une relation, il faut dans notre entité utiliser l'annotation @ORM\ manyToOne(), avec en parametre « targetEntity » et en valeur l'entité vers laquelle on veut créer une relation :

exemple dans l'entité Livres :

```
/**
```

```
* @ORM\ManyToOne(targetEntity="App\Entity\Author")
*/
private $author;
```

Pour faire une relation bi-directionnelle, on créé dans l'entité Author une propriété $book qui aura une relation OneToMany(), avec un parametre targetEntity qui prend en valeur l'entité qui contient la relation ManyToOne.
Il faut ensuite ajouter dans la relation ManyToOne (donc dans les Livres) le parametre inversedBy et dans la relation OneToMany le parametre mappedBy. En valeur on précise la propriété qui contient l'entité actuelle de l'autre côté de la relation. Par exemple dans les livres :

```
/**
* @ORM\ManyToOne(targetEntity=« App\Entity\Author », inversedBy=« book » )
*/
private $author;
```

et dans les auteurs :

```
/**
* @ORM\OneToMany(targetEntity=« App\Entity\Book », mappedBy=« author » )
*/
private $book;
```

Ensuite on re-génère les entité et on met à jour le schema de la dbb.

On a également les entités de type
: many-to-many : un invités peut participer à plusieurs évènements et un évènement peut accueillir plusieurs invités.

On a aussi les relation 1-1, plus rare.

Principe de cascade : quand on créé des associations, ça lie des entités entre elles. Et donc quand va ajouter des données, on va devoir spécifier les données nécessaires pour que ces liaisons / associations fonctionnent. C'est aussi le cas quand on détruit une entité, il faut gérer les entités qui sont liée : C'est le principe de la cascade, qu'on peut spécifier dans nos relations.

TPC : Insérer une donnée en bdd pour créer un auteur et un livre.
TPC : récupérer un auteur

TP : insérer plusieurs données dans les tables, créer le controlleur pour récupérer tous les livres, les afficher avec une view en grille.
TP : Afficher les infos relatives aux auteurs également
TP sup : créer un controleur qui affiche dans une page spéciale chaque auteur avec ses infos, et mettre le lien vers cette page dans chaque fiche de livre.

TP : Faire toutes les pages index, suppr, add etc des livres et auteurs

LES FORMULAIRES

Dans Symfony, on ne créé pas directement les formulaires en HTML. On va créer une classe PHP qui sera la représentation abstraite du formulaire, et on créera ensuite la vue qui correspond à ce formulaire.

Il reste à créer la view qui correspond à ce formulaire. Pour faire ça :

```
$form = $this->createForm(AuteurType::class, new Auteur);

$vars['form'] = $form->createView();
```

La méthode createForm créé un formulaire à partir de la représentation abstraite, et on y associe une entité.
Puis on créé la view à afficher de ce formulaire avec la méthode createView et on envoie

ça au fichier Twig.

Symfony prend le type de chacune des propriétés de l'entité et le transforme en type html adapté dans le formulaire. Par exemple une string sera transformée en TextType.

TP: Créer les formulaires des livres et auteurs en sortant les relations

Rendu des formulaires

On utilise la fonction twig form pour rendre un formulaire dans Twig. On lui passe en parametre la variable Twig qui contient le formulaire.

On peut utiliser dans la view form_start() qui va afficher la balise ouvrante et form_end() pour la balise fermante.

De manière implicite symfony appelle entre l'ouverture et fermeture la fonction form_rest() qui affiche le coeur de notre formulaire, et permet aussi d'afficher des éléments invisibles mais utiles comme les input hiden avec les token de sécurité.

Si on veut avoir encore plus de modularité :
- form_row(nom du champ), qui va contenir les fonctions form_label, form_widget et form_error. On peut aussi appeler ces fonctions de manière séparée.

On peut ajouter des classes css dans les « widgets » (qui créent les inputs en html) :
{{ form_widget(form.name, { 'attr': {'class': 'foo'} }) }}

TP: Créer les formulaires pour les auteurs et les livres

La classe abstractType contient deux méthodes principales :
- buildForm() pour construire le formulaire
- configureOptions() pour parametrer le formulaire (si besoin).

Dans configureOptions(), « data_class » nous permet d'associer l'entité au formulaire.
Dans buildForm(), pour chaque champs ajouté on peut passer plusieurs arguments :
- le type de champs html (DateType::class etc)

- un array d'options

Il faut aussi ajouter à la main le champs submit
->add('submit', SubmitType::class)

On peut aussi ajouter dans l'array d'options les labels associés aux champs

```
        ->add('dateMort', DateType::class,
                ['label' => 'Date de mort']
            )
```

des attributs :

```
        ->add('biographie', TextareaType::class,
            ['attr' =>
                ['placeholder' => 'merci de remplir la bio']
            ]
        )
```

Les champs de formulaires

Il existe de nombreux types de champs de formulaires : : CountryType, DateType, CheckboxType etc etc.

Options des champs :

Dans les champs de sélection (SelectType) on peut ajouter l'attribut 'preferred_choices' pour ajouter des choix préférés. Pour les champs nombre 'IntegerType), on peut choisir le nombre de décimales. Pour tous les champs on peut mettre 'required' avec en valeur true ou false.
Pour les champs de sélection, on peur définir la liste des choix avec 'choices' avec un array. On peut aussi modifier l'apparence avec 'expanded' et 'multiple'.

Champs d'entités :
Pour gérer les champs qui sont liées à une relation dans l'entité, il faut les déclarer comme EntityType.

On va ensuite devoir passer en parametres :
- class : avec en valeur la classe de l'entité qu'on veut pouvoir choisir
- query_builder (si besoin) : permet de n'afficher qu'une partie des résultats (par exemple si on ne veut pas afficher toutes les entités reliées mais seulement une partie).

Il faudra aussi définir le label à afficher grâce à :

```
'choice_label' => function($x) {

  return $x->getGenre();

}
```

TPC : création du form pour les auteurs
TP : création du form pour les livres

Utilisation

Quand le formulaire est créé dans le controleur, on peut, sur la même page, récupérer les données envoyées par le formulaire rempli.

La première étape est de vérifier que la requête contient une méthode POST (donc savoir si le formulaire a été soumis) grâce à « if ($request->isMethod('Post') ».
Ensuite il faut récupérer les données du formulaire et les relier à l'entité liée au formulaire grâce à « $form->handleRequest($request); »

Puis il faut vérifier que les données envoyées dans le formulaire sont valides (c'est à dire

qu'elles correspondent à ce qui est attendu, comme un nombre dans un champs nombre etc) avec « if ($form->isValid()) ».

Et enfin on sauvegarde les données du formulaire (stockées dans l'entité à l'étape 2) grâce à la méthode Persiste() puis Flush();

```
$em = $this->getDoctrine()->getManager();
$auteur = new Auteur();
$form = $this->createForm(AuteurType::class, $auteur);
if ($request->isMethod('Post')) {
    $form->handleRequest($request);
    if ($form->isValid()) {
        $em->persist($auteur);
        $em->flush();
    }
}
```

TPC : faire le form insert des auteurs

TP : faire le form insert des livres

Pour modifier un livre, la méthode est identique sauf que l'on passe un $id dans le controleur et qu'on utilise le repository en amont pour récupérer le livre (et c'est ce livre qu'on passe en entité du createForm, et non une entité vide).

```
{
    $em = $this->getDoctrine()->getManager();
    $rep = $em->getRepository('DocumentsBundle:Auteur');

    $auteur = $rep->find($id);
```

```
$form = $this->createForm(AuteurType::class, $auteur);


if ($request->isMethod('Post')) {
        $form->handleRequest($request);
        if ($form->isValid()) {
                $em->persist($auteur);
                $em->flush();
        }
}
```

TPC : faire le form update des auteurs

TP : faire le form update des auteurs

Pour supprimer, on a juste à utiliser la méthode remove de l'entité manager sur une enttié.

Pour générer tout d'un coup, on a une commande crud qui permet de créer d'un coup les routes, controller, form etc pour nos entités.

TP : Faire crud pour les livres:
Liste tous les livres sur une page. Pour chaque auteur : bouton de modif et suppression. Bouton pour ajouter un auteur redirige vers un form.

Validations

Pour valider les données de formulaires, le plus simple est d'utiliser les annotations.

On a un composant qui s'appelle Validator et qui veille à ce que des contraintes liées à une entité sont respectées. Les contraintes permettent de définir les données qu'on attends et celle qu'on refuse, pour chaque propriété d'une entité. Une fois définie ces contraintes seront utilisées par Symfony pour valider ou non chaque champs d'un formulaire envoyé (car un formulaire est lié à une entité).

Pour définir une contrainte, on annote le fichier d'entité avec la classe constraints. Il existe plusieurs contraintes : Blank, Email, Url, Range, Length etc :

```
* @Assert\Length(
*     min = 2,
*     max = 100,
*     minMessage = 'Le titre doit au moins comporter 2 caractères',
*     maxMessage = 'Le titre ne doit pas comporter plus de 100 caractères'
* )
```

Attention à bien importer la classe constraints et à ne pas utiliser de simple quotes.

TP : ajouter des contraintes sur tous les champs du form des livres et auteur

Formulaire enfant :

On peut ajouter un formulaire enfant dans un formulaire :

Par exemple, quand deux entités sont liées, dans la classe de formulaire Type, au lieu d'utiliser une EntityType, on peut passe directement un formulaire.

Message Flash

Les message flash sont des messages destinés à être affichés une seule fois sur une page, après qu'une action ait été effectuée.
Pour créer un message flash après une action utilisateur (comme la création d'un nouvel item en base de données), on peut utiliser dans le controleur :

$this->addFlash('warning', 'message erreur');

Pour afficher un message flash dans un fichier twig :

```
{% for message in app.flashes('notice') %}
    <div class="flash-notice">
        {{ message }}
    </div>
{% endfor %}
```

Ajouter des images

TP: En suivant la documentation de Symfony (***https://symfony.com/doc/current/controller/upload_file.html***) ajouter des images aux livres, en utilisant une propriété image dans l'entité.

Sécuriser l'application

La création d'un espace administrateur peut être faite sans utiliser de bundles. Mais nous allons utiliser ici Fos User pour tester l'installation d'un bundle.

TPC : sécuriser toutes les parties admin de son application grâce à FOS User (https://vfac.fr/blog/how-install-fosuserbundle-with-symfony-4)

Mise en production

http://david-robert.fr/articles/view/deployer-symfony-vps

Lu et approuvé par Cecile Salon

« Ma parole c'est le meilleur support de cours que j'ai jamais vu » : Marc Levy

« It helped me writing season 1, 2 and 3 » Georges Martin

« J'ai pleuré » Almodovar