**Symfony**

Table des matières

[Genéralité et installation 3](#_Toc77674363)

[Composer 3](#_Toc77674364)

[Initialisation d’un dossier symfony 3](#_Toc77674365)

[Php storm 3](#_Toc77674366)

[Hello world 4](#_Toc77674367)

[Création du routeur 4](#_Toc77674368)

[Affichage avec twig 4](#_Toc77674369)

[Twig 5](#_Toc77674370)

[Les balises twig 5](#_Toc77674371)

[Les filtres twig 5](#_Toc77674372)

[Bloc et héritage 6](#_Toc77674373)

[Gestion des assets et des URLs 6](#_Toc77674374)

[Passer des variables à la vue 7](#_Toc77674375)

[Attaque XSS sous twig 7](#_Toc77674376)

[Routage et controleurs 7](#_Toc77674377)

[Le parametre URL 8](#_Toc77674378)

[Les routes : requirements et autres conditions 8](#_Toc77674379)

[Quelques commandes de débogage pour les routes 8](#_Toc77674380)

[Les controleurs 8](#_Toc77674381)

[Préfixe sur les controleurs 9](#_Toc77674382)

[créer un nouveau controlleur en commande 9](#_Toc77674383)

[Débogage 9](#_Toc77674384)

[Doctrine 9](#_Toc77674385)

[Les entités 10](#_Toc77674386)

[EntityManager 10](#_Toc77674387)

[Le repository 11](#_Toc77674388)

[Requetes personnalisé 11](#_Toc77674389)

[Requête non préparée 11](#_Toc77674390)

[Requête préparée 12](#_Toc77674391)

[Formulaire 12](#_Toc77674392)

[Création d’un formulaire 12](#_Toc77674393)

[Affichage de notre formulaire 13](#_Toc77674394)

[Traitement du formulaire 14](#_Toc77674395)

[Validation des données de formulaire 15](#_Toc77674396)

[Comment faire ? 15](#_Toc77674397)

[Sécurité : les attaques CSRF 16](#_Toc77674398)

[Les relations entre entités 16](#_Toc77674399)

[Type de relation 16](#_Toc77674400)

[Les relations avec doctrine 16](#_Toc77674401)

[Créer une relation 16](#_Toc77674402)

[Récupérer une entité associée à une autre 17](#_Toc77674403)

[Manipuler les entités associées 17](#_Toc77674404)

[Exemple d’enregistrement d’un post et d’un commentaire associé 17](#_Toc77674405)

[Exemple de suppression 18](#_Toc77674406)

# Genéralité et installation

Symfony est un Framework qui permet d’avoir une structure spécifique afin d’avoir une organisation cohérente et qui permet un travail en équipe, évite de « réinventer la roue » et évite quelques problèmes de sécurité.

Il existe 2 versions possible de symfony sur laquel travaillé :

* le skeletons (version minimaliste et personnalisable)
* le website-skeleton (version complète et personnalisable)

## Composer

Composer permet en ligne de commande d’initialiser un dossier et d’installer des dépendances. Il faut installer commander sur le site officiel mais aussi cmder pour avoir un interpréteur de ligne de commande mieux que celui de windows.

## Initialisation d’un dossier symfony

Une image contenant texte

Description générée automatiquement

Puis dans le dossier que l’on viens de créer :



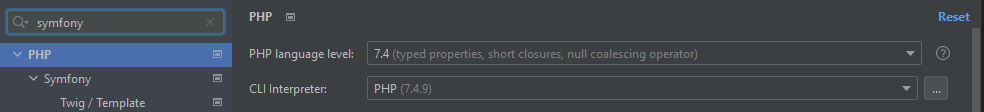
*Pour vérifier sa bonne implémentation, présence d’un fichier .htaccess dans /public.*

## Php storm

Une image contenant texte

Description générée automatiquementSur **PHPStorm** il faut installer le plugin symfony, l’activer quand on lance un projet symfony (une pop-up en bas a droite) et configurer dans les paramètre tel que

* enable plugin doit être coché
* Décoché les code folding (qui raccourcis le code)
* Ajouter le dossier /.idea/ dans gitignore



# Hello world

Si problème de drivers, dans fichier .env commenter postgres et décommenter mysql avec les bon id et donner un nom de bdd



Puis taper la commande dans le terminal

Une image contenant texte, orange

Description générée automatiquement

## Création du routeur

Dans src>controller créer un fichier nommé MainController

Une image contenant texte

Description générée automatiquement

## Affichage avec twig

Dans templates créer un dossier main>home.html.twig

Une image contenant texte

Description générée automatiquement

# Twig

Twig permet en plus d’écrire du html classique d’y ajouter du code PHP. Twig est minimaliste et performant

Il existe 3 délimiteurs en twig

* **{# … #}** : pour écrire des commentaires
* **{{ … }}** : appeler double moustache permet de faire un echo
* **{% … %}** : pour exécuter du code (ce n’est pas du code php mais des balises twig)

Pour concaténer dans twig il faut utiliser le ~

## Les balises twig

Une image contenant texte

Description générée automatiquementUne image contenant texte

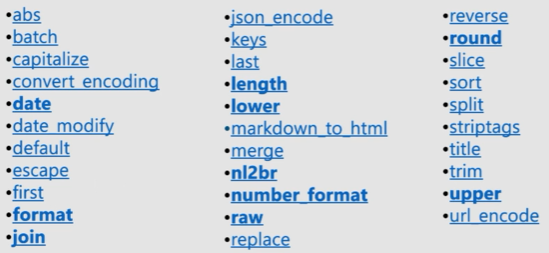
Description générée automatiquement



## Les filtres twig

Une image contenant texte

Description générée automatiquementLes filtres permettent de modifier des variables, pour les utiliser on ajoute un pipe | après la variable



## Bloc et héritage

Avec twig nous pouvons créer un « gabarit » de page avec des bloc qu’ont laisse vide (ou une valeur par défaut).

Une image contenant texte

Description générée automatiquement

Et maintenant notre page home ressemble à ça

Une image contenant texte

Description générée automatiquement

## Gestion des assets et des URLs

Pour avoir une gestion des url (afin d’éviter les problème avec les chemins relatif qui ne trouverons pas toujours le bonne emplacement) on utilise **asset(‘folder/rss’)** qui permet de rediriger vers le dossier /public



Et pour les url de page on utilise **path(‘name\_route’)**

Une image contenant texte

Description générée automatiquement

Et pour passer un paramètre en plus on ajoute un objet tel que {‘name’ : value}



## Passer des variables à la vue

Pour passer des variable à la vue, il faut fournir en 2em arguments de la méthode render un tableau

Une image contenant texte

Description générée automatiquement

On peut aussi utiliser la méthode compact, qui ira chercher les variable qui porte le nom entre « » tel que :

Une image contenant texte

Description générée automatiquement

Il ne reste plus qu’a l’utiliser dans notre twig



## Attaque XSS sous twig

Par défaut twig « protège » des faille XSS les echo que l’ont fait avec **{{ … }}** par défaut. Dans certain cas on pourrait avoir envie de garder la lecture des balises en utilisant **raw**, **stripstags** si on fait confiance au data que l’on affiche, sinon si on est entre les deux on pourrait utiliser une bibliothèque tel que **HTMLPurifier** qui va limite l’autorisation à certaine balise.

Exemple :

# Routage et controleurs

Le routage symfony permet entre autres de centralisé les accès au URL avec le MainController, de pouvoir avoir des url définies plus simplement.

Pour effectuer le routage il faut faire un commentaire avant notre fonction qui commence toujours par **/\*\*** et dans le commentaire **@Route** (penser à importer la class correspondante). Le premier paramètre est le pattern URL.

Une image contenant texte

Description générée automatiquement

## Le parametre URL

Une image contenant texte

Description générée automatiquementIl doit être unique et en général définie des le début de l’application. Dans une URL nous pouvons avoir une partie variable (notamment avec les **GET**) c’est ce qu’ont appel des **jokers**.

Une image contenant texte

Description générée automatiquementPour éviter certains conflits il est une bonne pratique de définie via une expression rationnelle le résultat attendus, par exemple ici nous acceptons 1 ou plusieurs chiffres

Une image contenant texte

Description générée automatiquementOn peut également définir une valeur par défaut si l’URL n’en as pas

## Les routes : requirements et autres conditions

Nous pouvons avoir des adresse identique en URL qui rendront un résultat différent suivant le type de requête (**GET,POST**) ce qui sera très utile pour construire une API REST.

Une image contenant texte

Description générée automatiquementLe nom des routes est très fortement recommandé, elles permettent de générer les **URL** correspondant aux routes, par convention on les nomme avec le nom du contrôleur (ici main) et le nom de la méthode en **snake\_case**.

## Quelques commandes de débogage pour les routes

Une image contenant texte

Description générée automatiquement

## Les controleurs

Quand on créer un contrôleur il faut qu’il hérite de **AbstractControlle**r, ce qui permettra d’accéder a un tas de fonctionnalité. Il faut essayer de limiter nos contrôleurs à une dizaine d’URL maximum, chaque méthode du contrôleur représente une page. Grace au contrôleur nous avons un accès facile à **HttpFoundation\Request** et nous devons obligatoirement retourner un **HttpFoundation\Response.**

**Exemple de retour possible :**



## Préfixe sur les controleurs

Une image contenant texte

Description générée automatiquement

## créer un nouveau controlleur en commande

Une image contenant texte

Description générée automatiquement

## Débogage

Une image contenant texte

Description générée automatiquement

Avec **dump** on verra dans la barre du bas afin de voir le détail de la variable.

**dd()** permet de le voir directement dans la page

Une image contenant texte

Description générée automatiquementEn bas du site est disponible une barre pour avoir des infos et lors du click sur la barre on as accès au profiler qui servira pour le débogage, les formulaire, la sécurité et les requête en base de données.

# Doctrine

Doctrine est un ORM (couche d’abstraction à la base de données) pour PHP. Il permet entre autres de dialogué avec la base de données de manière simple parfait même sans écrire une ligne de code SQL, les commandes les plus utilisé (select \*, select by id…) sont implémenté et s’use facilement.

Pour la configuration il y as **/.env** (pour le modèle) et **/.env.local** sur lequel on travaillera, l’intérêt de différencier le modèle du local est pour le travail sur notre station de travail quand on est plusieurs sur un projet (et que probablement nous n’avons pas les même droit) En revanche lors de la création de la base de données il faudra reconfigurer dans phpMyAdmin l’encode en utf8\_general\_ci

## Les entités

Ce sont les classes PHP qui vont représenter nos données (plutôt que des tableaux associatifs qui sont moins fiable). On va utiliser des **annotations** afin de donner des directives à **doctrine** qui se chargera de s’occuper de la communication avec la base de données. Ces classes sont nommées des **entités** et sont le cœur de doctrine. L’ensemble de ces **entités** est nommé **schéma**.

Pour créer une nouvelle entité :

Une image contenant texte

Description générée automatiquement

Ensuite l’id est généré automatiquement (ne pas le spécifier donc), il faudra spécifier le nom et le type des colonnes que l’on souhaite ajouter, une fois finis on appuis sur entrer et le fichier sera généré.

*Une image contenant texte, horloge, périphérique

Description générée automatiquement*

*Pour afficher les types accepter faite****?*** *puis entrer, et une liste apparaitra*

Une fois que tout est fait, il faut faire une commande afin que doctrine créer la table

**

## EntityManager

Une image contenant texte

Description générée automatiquementL’**entityManager** de doctrine permet de gérer les requêtes aux bases de données, la classe permet de gérer le **CRUD** mais pas le **R** **Select** (ce sont les repository qui le font) ce sera toujours le même **entityManger** utiliser pour toutes les entités.

Afin de pouvoir interagir en base de données avec doctrine il faut ajouter dans un paramètre de fonction un **EntityManagerInterface**

Une image contenant texte

Description générée automatiquement

Ensuite on créer une instance d’une entité et on l’hydrate (fournis les données)

Une image contenant texte

Description générée automatiquementEt ensuite on peut utiliser notre **EntityManagerInterface** afin d’interagir avec la base de données.

* Persist : pour sauvegarder en base de données (create/update)
* Remove : pour supprimer (remove)
* Une image contenant texte

  Description générée automatiquementFlush : pour commit les changements

## Le repository

On disposera d’un **repository** par entité, ils sont générer quand on génère une entité avec le entity:make. Ils permettent de faire des requêtes de type select.

**Méthodes principales du repository :**

Une image contenant texte

Description générée automatiquementfindOnByTitle viendra automatiquement chercher si une colonne porte le nom title, mais préférer findOneBy()

Pour utiliser un **repository** dans la fonction sur lequel on veut récupérer les données on donne en paramètre la classe repository associé a notre demande, ici par exemple SerieRepository pour lequel on voudra récupérer l’ensemble des données

Une image contenant texte

Description générée automatiquement

On peut utiliser des filtres avec un findBy, par exemple ici on récupère les 30 entrée classé par popularity (si exæquo par vote):



## Requetes personnalisé

Pour les requêtes plus spécifiques on va utiliser **DQL** (doctrine query langaue), qui au lieu de faire des requêtes au table on fera des requêtes au class et **QueryBuilder** qui nous permet de bâtir des requêtes. Il faudra choisir entre l’un des deux.

### Requête non préparée

Une image contenant texte, capture d’écran, personne

Description générée automatiquement**Exemple de DQL**

* On fait un lien vers notre classe au lieu de la table (App\Entity\Product)
* Il **FAUT** précéder les champs du nom de la classe
* Il n’existe pas de **LIMIT** dans la requête, ici nous utilisons **setMaxResults()**

**Exemple de QueryBuilder**

* Une image contenant texte

  Description générée automatiquementOn créer un **QueryBuilder** en lui fournissant un alias (ici ‘p’)
* Le **QueryBuilder**, par défaut va chercher dans la classe liée au **repository**
* Ensuite on enchaine les clause where avec ->**andWhere(**name > qte**)**
* On utilise **setMaxResults()** en guise de **LIMIT**
* Puis on exécute la requête avec **getQuery()** et **getResult()**

### Requête préparée

Dans ces exemples nous recevons une data d’un champ de formulaire (un filtre) nommé minRating

**Exemple avec DQL**

Une image contenant texte

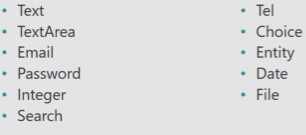
Description générée automatiquementIci on doit faire un if() et concaténé un AND … pour ajouter une clause dans le WHERE

**Exemple avec QueryBuilder**

Une image contenant texte

Description générée automatiquementIci on fait un if() et si on as bien une valeur dans $minRating on ajoute dans le clause **WHERE** (andWhere) puis on **BIND** la valeur (setParameter)

# Formulaire

Symfony à créer un composant **Form**, qui permet d’automatiser la validation des données et la génération du HTML. Un formulaire est associé à une entité (ou plusieurs tables sans problèmes), dès la soumission du formulaire Symfony peut injecter directement les données etc.

Les formulaires disposent de plus d’une trentaine de types de champs

Une image contenant texte

Description générée automatiquementPar exemple entity permet d’avoir une liste deroulante dont le contenu pourra être celui d’une table en base de données

On peut ajouter des options sur nos type de champs

## Création d’un formulaire

Pour créer une classe de formulaire on va utiliser la commande

Une image contenant texte

Description générée automatiquementEnsuite on doit donner un nom au formulaire, toujours faire suivre le nom du formulaire par Type comme ci-dessous

Et ensuite on lui fournit le nom de l’entité lié 

Ce qui créera un nouveau fichier dans From > SerieType que l’ont pourra modifier, comme par exemple :

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquementsi un attribut est une entité, il faut le spécifier dans la création du Form sinon on aura une erreur quand on essayera de l’utiliser. Ici on créer une liste déroulante avec des serie.name

Lors de l’import de type bien choisir la bonne classe :

## Affichage de notre formulaire

Tout d’abord on envois notre formulaire à la vu (twig)

Une image contenant texte

Description générée automatiquementne pas oublier d’ajouter une Request

use Symfony\Component\HttpFoundation\Request;

Une image contenant texte, orange

Description générée automatiquementEt dans twig, la manière la plus simple :

Une image contenant texte

Description générée automatiquement

Le souci est que pour modifier notre formulaire comme ajouter un bouton on devra décomposer notre formulaire, le plus simple est de faire ceci :

Pour décomposer encore plus loin on peut spécifier l’affichage de chaque input avec les label et message :

Une image contenant texte

Description générée automatiquement

## Traitement du formulaire

Le traitement du formulaire se fera dans la même méthode que celle de l’affichage. On test d’abord si le formulaire est bien soumis, puis on récupère les données on en fait quelque chose, on redirige vers une autre page et on affiche un message à l’utilisateur.

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

Et pour l’affichage des message flash avec twig (typiquement placer entre header et body du template de base)

## Validation des données de formulaire

Les validations peuvent se faire de différentes manières et c’est plutôt une configuration que du code a proprement parlé. Cela peut être fait en XML, YAML, PHP ou par annotations (ce qui est le plus souvent le cas). On applique des contraintes qui sont des règles de validation, ils en existent plus de 40 de base et nous pouvons en créer nous même

Quelques contraintes existantes :



### Comment faire ?

Pour appliquer les contraintes dans l’entité par annotation, il faut tout d’abord faire attention à ne pas oublier de donner un alias à la classe **Constrains**

Une image contenant texte

Description générée automatiquement

Puis pour chaque attribut on doit spécifier les contraintes et les messages d’erreur associé

Une image contenant texte

Description générée automatiquement

Et enfin on vérifie les contraintes avant d’ajouter les données en DB

Une image contenant texte

Description générée automatiquement

## Sécurité : les attaques CSRF

Cette attaque se pratique quand on formulaire est envoyé depuis une autre source que notre site, pour s’en protéger il faut ajouter un champ caché dont nous seul connaissons la valeur. Symfony le fait automatiquement avec le composant Form 😊

# Les relations entre entités

## Type de relation

Il existe plusieurs types de relations :

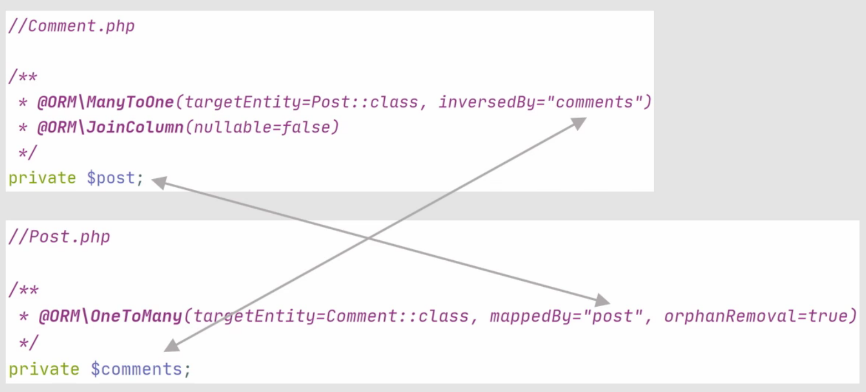
|  |  |  |
| --- | --- | --- |
| **OneToMany / ManytToOne** | **ManyToMany** | **OneToOne** |

Une relation ManyToMany implique qu’il existe une table supplémentaire afin d’enregistrer les relations entres les entités

## Les relations avec doctrine

Les relations avec doctrines sont des relations entre des entité, il faut faire abstraction des tables que doctrine utilise, **on n’utilise jamais les ID** pour créer les relations.

Une image contenant texte

Description générée automatiquementUne relation se fait par annotations tel que

## Créer une relation

Une image contenant texte

Description générée automatiquementTout d’abord on créer une entity  comme d’habitude.

Ensuite on refait une deuxième fois la même commande. Cette fois ci on ajoute la propriété qui fera la liaison de la table actuelle avec celle avec qui on veut faire une relation, en lui fournissant le type **relation** et le nom de l’**entity** à lequel on la relie.

* On lui indique si on veut faire une [ManyToOne, OneToMany, ManyToMany, OneToOne]
* Si le champ peut être null
* Si on veut créer une association bidirectionnelle (ici si on peut accéder au seasons depuis serie)
* Une image contenant texte

  Description générée automatiquementLe nom du nouveau champ

Il ne reste plus qu’a mettre à jour la base de données, pour cela on fait un 

Mais si on veut voir avant la requête qui sera exécuter on peut faire un



## Récupérer une entité associée à une autre

Grace a doctrine les requêtes SQL seront fait seulement si besoin, par exemple si on a un blog avec des articles et des commentaires associés, tant qu’on ne demande pas à accéder au commentaire (que ce sois directement dans **twig** ou ailleurs) doctrine ne fera pas cette requête supplémentaire pour les avoir, c’est ce qu’ont appel le **lazy loading**. Dans le cas d’une page d’un article avec les commentaires cela fera 2 requêtes, en revanche si on affiche une page avec plusieurs articles et les commentaires associé, imaginons qu’ont à 60 articles, doctrine fera donc 120 requêtes (les article et les commentaire) ce qui n’est pas optimal. Dans ce cas-là il faudra faire une jointure manuelle qui sera fait dans le repository.

Une image contenant texte

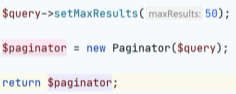
Description générée automatiquementPour faire une requête de jointure à la main on fera quelque chose comme :

Ou le premier argument est la propriété que l’on veut récupérer, et en deuxième un alias que l’on donne. Ne pas oublier le **->addSelect** sinon les résultats ne sont pas retournés.

Une image contenant texte

Description générée automatiquementOn peut rencontrer un souci avec le **setMaxResult** quand on affiche les résultat, par exemple ici on limite à 50, mais il fera cette limite sur l’attribut que l’ont viens d’ajouter (ici les seasons).

Pour remédier à ce problème on peut utiliser un **Paginator**



## Manipuler les entités associées

### Exemple d’enregistrement d’un post et d’un commentaire associé

Une image contenant texte

Description générée automatiquement

*NB : on fournit un $post et nom un id de post à $comment, ce sera doctrine qui se chargera des liaisons avec les ID*

### Exemple de suppression

*Une image contenant texte

Description générée automatiquement*

Ici ça ne fonctionnera pas, car nous essayons de supprimer un post qui as des commentaires qui lui sont associés. Il faudrait alors supprimer d’abord les commentaires, puis le post mais il y as mieux…

Pour le faire il faudra le spécifier dans les annotations avec cascade={« remove »}

