



## Avant de commencer

- Présentations
- Tour de table
- Horaires de formation
- Organisation du cours
- Outillage

## Introduction aux frameworks

- Développer en équipe
- L'outillage d'une équipe de développeurs
- Objectifs d'un framework

## Introduction à Symfony

- Historique du framework
- Pré-requis
- Composer
- Le squelette de projet
- Dépendances incluses

## Premier projet Symfony

- Lancer le projet
- À la racine du projet
- MVC et requêtes HTTP
- Création d'un contrôleur
- Template de page et style
- Action du contrôleur
- Configuration des routes

## Gestion des données avec Doctrine

- Pré-requis
- Création d'une entité
- Migration de base de données
- Fixtures

## CRUD : créer une entité

- Introduction aux formulaires
- Création d'un FormType
- Affichage de formulaire
- Amélioration de l'affichage
- Récupération des données
- Validation et sauvegarde de données

## CRUD : lire, éditer, supprimer

- Requête des entités
- Créer une requête personnalisée
- Édition d'une entité
- Suppression d'une entité

## Relations entre entités

- Configuration d'une relation Doctrine
- Fixtures et liens entre entités

## Gestion des messages

- Locales
- Localisation des formulaires
- Messages flash

## Tests fonctionnels et unitaires

- Tests unitaires
- Data providers et dépendances
- Création de doublures
- Tests fonctionnels

## Sécurité

- Création d'utilisateur
- Formulaire de login
- Restriction d'accès
- Restriction de template
- Déconnexion
- Voters

## Déploiement

- Installation sur le serveur de prod

## Bundles populaires

- Webpack Encore

- Easy Admin Bundle
- FOS REST Bundle
- Hautelook Alice Bundle
- Vich Uploader
- Doctrine Extensions
- FOS User Bundle

## Différences avec Symfony 3

- Organisation du projet
- Autres différences

## Ressources

# PHP599-F : Symfony initiation et approfondissement

*Durée : 5 jours*

*Auteur : K. Thommy*

*Version Symfony : 4.2*

# Avant de commencer

## Présentations

Kevin Thommy, formations web / Symfony.

- parcours varié : graphisme, intégration, développement
- promotion POE Dawan de juin 2017
- je donne ponctuellement des formation sur mes spécialités

## Tour de table

Par personne :

- nom, parcours
- facilités et difficultés sur le cursus
- choses à revoir
- attentes par rapport au cours

## Horaires de formation

- horaires de repas et de pause habituels
- y a-t-il des gens qui doivent partir plus tôt vendredi ?
- attention aux absences par rapport à la POE

## Organisation du cours

Je propose un cours très orienté pratique :

- en groupe ça nous laisse plus de temps pour de la correction individuelle
- Symfony est un outil qui avance vite, les livres et cours ont souvent du retard

- c'est un outil de code : plus on pratique, mieux on comprend

Le cours se portera sur le framework Symfony mais aussi sur les outils qui l'accompagnent quand c'est nécessaire.

On va suivre un projet toute la semaine : le site d'un journal, avec parution d'articles, gestion des auteurs, etc. Ça nous permettra d'être dans le concret !

## Outillage

Je vous laisse utiliser votre IDE préféré, si vous n'en avez pas l'éditeur Atom avec quelques plugins est une solution viable pour le temps de la formation.

# Introduction aux frameworks

## Développer en équipe

- séparer les tâches
- synchroniser le travail
- se retrouver dans le travail des autres (contributions)
- créer des conditions identiques de développement
- déployer une version testable par tout le monde
- éviter les régressions

## L'outillage d'une équipe de développeurs

- issue tracker (gestionnaire de tickets)
- gestionnaire de versions (vcs, version-control system)
- gestionnaire de dépendances
- tests unitaires et fonctionnels
- déploiement automatisé / intégration continue
- documentation auto-générée
- framework (ou *cadriciel*)

Le framework est donc un élément d'un écosystème de travail qui facilite le développement logiciel en équipe.

## Objectifs d'un framework

- organiser le code selon des règles communes approuvées par des spécialistes

- isoler les responsabilités de chaque morceau de code
- rendre le travail commun facilement extensible
- gagner du temps pour le code métier, la spécification, la documentation et les tests

Un framework n'est pas une bibliothèque : ne pas confondre une architecture de code (framework) avec une collection de fonctionnalités (bibliothèque).

# Introduction à Symfony



Symfony est l'un des frameworks PHP les plus populaires. Il dispose d'une excellente documentation disponible sur [symfony.com](https://symfony.com) (on s'y réfèrera sûrement pendant le cours).

## Historique du framework

- 2005 : Fabien Potentier rend open-source le framework PHP de son entreprise sous la license MIT. Il le nomme Symfony pour garder l'ancien logo (Sensio Framework).
- 2007-2008 : le logiciel se stabilise en ce qu'on appelle couramment *Symfony 1*.
- 2011 : *Symfony 2* est une réécriture complète du framework. La structure actuelle de l'outil est mise en place.
- 2015 : le rythme de parution des versions est figé et des mécanismes mis en place pour faciliter la migration.
- Fin 2017 : la version majeure actuelle, *Symfony 4*, est livrée.

Symfony c'est aussi les *Symfony Components*, une série de bibliothèques créées par les développeurs de Symfony et utilisées par nombre de projets PHP importants : Drupal 8, Joomla, Magento, Laravel, Google API, Facebook Ads, Composer, phpBB, PHPMYAdmin ...

## Pré-requis

Pour utiliser Symfony on a besoin au minimum de :

- PHP 7.1.3
- Composer

On peut aussi, après l'installation de Symfony, passer par le *Requirements Checker* qui



vérifie plus en détail les éléments nécessaires.

### Atelier : installation de php

Sur Ubuntu, la commande d'installation est `apt install`. Il faut passer en *mode super utilisateur* (et entrer le bon mot de passe) pour pouvoir installer quelque chose :

```
sudo apt install php
```

## Composer

Composer est le gestionnaire de dépendances de Symfony et beaucoup d'autres projets ensemble. C'est avec lui qu'on construit un projet Symfony.

Les avantages d'un gestionnaire de dépendances :

- on ne stocke pas les dépendances dans le projet (plus léger à synchroniser)
- on peut reconstruire un projet à partir d'un résumé de ses dépendances ou en créer un à partir d'un squelette
- il est capable de décider ce dont on a besoin en plus pour utiliser un outil ou une bibliothèque

### Atelier : installation de Composer

Le plus simple est d'aller sur le site [getcomposer.org](https://getcomposer.org) et de suivre les instructions.

## Le squelette de projet

Chaque démarrage Symfony se fait à partir d'un squelette, un projet nu qui comporte l'essentiel pour construire un site. Les deux principaux squelettes disponibles sont :

- `symfony/website-skeleton`, un squelette complet qui comprend tous les outils nécessaires au développement d'une application web
- `symfony/skeleton` qui comprend le minimum de dépendances pour faire

fonctionner le framework

Le démarrage d'un projet est assuré par Composer.

 Atelier : démarrage d'un projet Symfony avec Composer

On va utiliser le squelette pour site web et une commande Composer appelée **create-project** :

```
composer create-project symfony/website-skeleton mon-projet
```

Dépendances incluses

- **symfony/flex** : un plugin Composer qui permet de configurer automatiquement les dépendances qu'on ajoute
- **symfony/console** : la ligne de commande symfony (on l'utilise beaucoup)
- **symfony/profiler-pack** et **symfony/debug-pack** : outils de debug et profilage de code dans le navigateur
- **symfony/web-server-bundle** : un serveur simple pour lancer son application en développement
- **symfony/maker-bundle** : un outil pour automatiser la création des fichiers courants du framework
- **doctrine** : un ORM (gestionnaire de base de données) capable de gérer une base tout seul à partir d'une configuration
- **swiftmailer** : envoi de mails
- **twig** : un moteur de templates
- **phpunit** : tests unitaires et fonctionnels

... et bien d'autres !

On remarquera que dans les dépendances un certain nombre sont des « bundles », c'est à dire qu'elles ont un format spécial reconnu par Symfony qui permet de les configurer dans l'application.

# Premier projet Symfony

## Lancer le projet

Testons d'abord que notre application s'est bien installée en lançant le serveur. Pour ça on va se servir de la *console Symfony* citée plus haut :

```
cd mon-projet  
bin/console server:start
```

On peut se rendre à l'adresse indiquée et parcourir le site et le profiler.

## À la racine du projet

Les dossiers :

- **bin** : les outils en ligne de commande, principalement la console
- **config** : la configuration de l'application
- **public** : le dossier public, là où est lancé le serveur
- **src** : le code php de l'application
- **templates** : les templates (ou maquettes) pour les pages, etc.
- **tests** : là où on range les tests de phpunit
- **translations** : les différentes locales et traductions
- **var** : fichiers temporaires et journaux
- **vendor** : dépendances composer

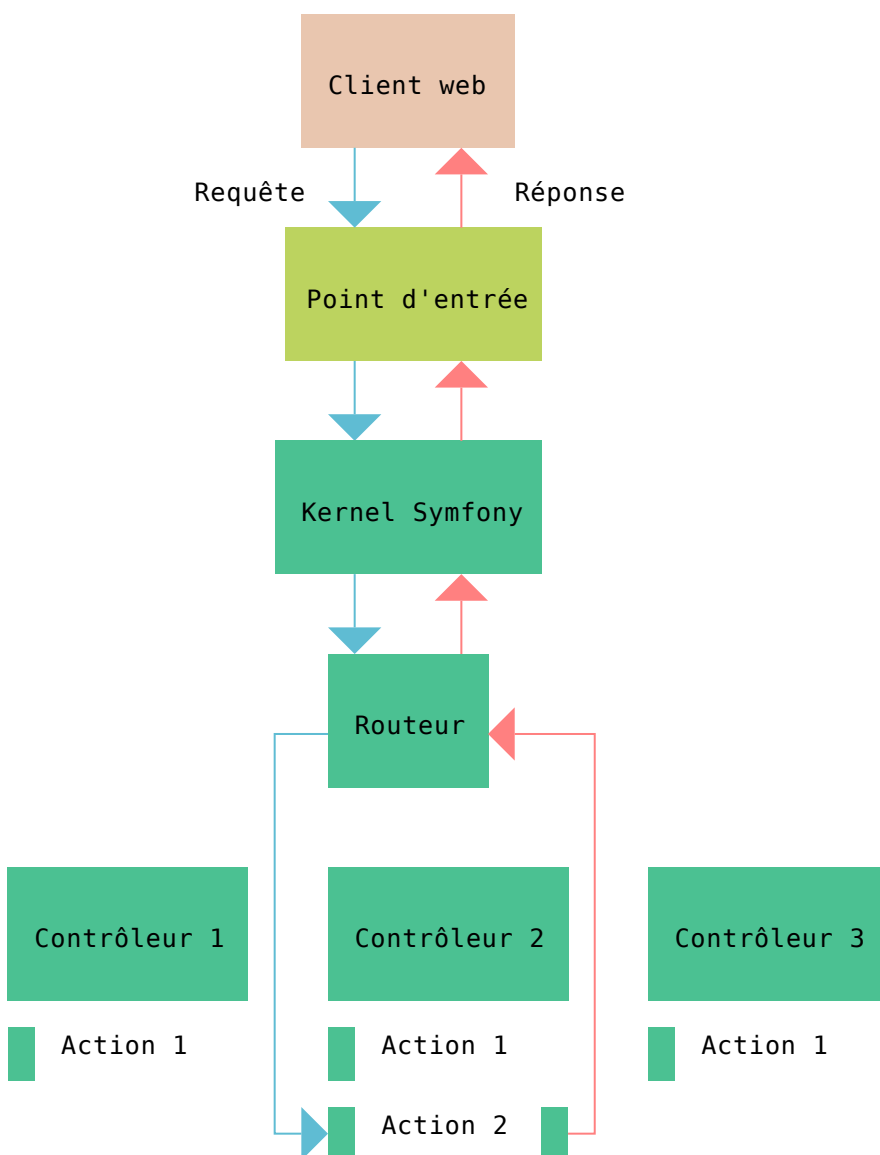
Les fichiers :

- **composer.json** et **composer.lock** : configuration composer, .json pour la configuration générale et .lock pour l'état spécifique de chaque dépendance

- `.env` et `.env.test` : variables d'environnement pour le développement et le test
- `.gitignore` : liste des fichiers à ne pas versionner
- `phpunit.xml.dist` : configuration phpunit
- `symfony.lock` : état spécifique des recettes flex

## MVC et requêtes HTTP

Symfony est basé sur une architecture MVC (Modèle, Vue, Contrôleur). En tant qu'application web, elle répond à des requêtes :



## Création d'un contrôleur

Par défaut la page servie est une 404, il est temps de créer une page d'accueil. Pour ça il nous faut un contrôleur dans lequel on définira l'action d'affichage de page d'accueil.

On va introduire le maker :

```
bin/console make:controller
```

Si par exemple on l'appelle `HomeController`, le maker va créer les fichiers nécessaires et rendre l'action par défaut accessible à l'url `/home`.

## Template de page et style

Les templates sont basés sur Twig lorsqu'on utilise le squelette de site web. C'est un moteur développé par les auteurs de Symfony.

Une syntaxe simple :

- affichage de variable `{{ my_var }}`
- instructions et boucles `{% if my_boolean %}...{% endif %}`
- filtres `{{ my_title | capitalise }}`  
et appel de fonctions `{{ form(my_form) }}`

Les assets (css, images, js, ...) sont rangés dans le dossier public et appelés dans Twig par la fonction `asset('css/my-style.css')`.

Les liens vers d'autres pages sont créés par la fonction `path('my_route')` qui prend un nom de route et optionnellement des paramètres.

 Atelier : personnaliser la "home"

Aller voir la documentation disponible sur [twig.symfony.com](https://twig.symfony.com) et modifier le fichier `templates/home/index.html.twig`.

## Action du contrôleur

Le rendu du template est géré par le contrôleur. Il est l'élément central qui manipule le

reste. Il reçoit une requête et renvoie une réponse.

- on respecte les conventions de nommage : `{Something}Controller`
- des méthodes accessibles à tous les contrôleurs qui étendent `AbstractController`
  - `render()` pour transformer un template en réponse HTTP
  - `redirectToRoute()` qui renvoie une redirection
  - `createFormBuilder()` pour aider à la création de formulaires
  - `generateUrl()` génère une url complète à partir d'une route et de paramètres
  - `trans()` pour traduire des messages
  - etc.

 Atelier : passer des variables au template

La fonction `render()` prend en arguments un template et un tableau de variables. On peut lui passer ce qu'on veut et l'afficher dans le template.

## Configuration des routes

De base la configuration des routes est dans `config/` au format YAML. Il y en a plusieurs qui se surchargent ou se complètent :

- `config/routes.yml` : fichier générique, plutôt pour des routes fixes
- `config/routes/annotations.yml` : liste les dossier dans lesquels il y a de la *configuration par annotations*
- `config/routes/dev/*` : configuration spécifique à l'environnement de dev (ni test, ni prod)

La configuration créée par le maker est par défaut une annotation au-dessus de l'action du contrôleur.

```
class HomeController extends AbstractController
{
```

```
/**  
 * @Route("/home", name="home")  
 */  
public function index()  
{  
}
```




# Gestion des données avec Doctrine

Doctrine est l'ORM embarqué avec le squelette de site web. Il permet de créer des entités (modèle PHP) qui seront mises en lien automatiquement avec la base de données au travers d'une configuration.

## Pré-requis

Pour utiliser Doctrine il faut évidemment une base de données et une configuration de l'environnement avec la base, l'utilisateur et le mot de passe renseignés.

 Atelier : installation de MySQL et configuration de l'environnement

```
sudo apt install mysql-server
```

Il est possible que des étapes supplémentaires soient nécessaires sur des versions récentes de MySQL.

Copier le `.env` vers `.env.local`, le modifier et lancer la création de la base de données :

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

## Création d'une entité


Le maker permet la création rapide d'entités.

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

La configuration de l'entité passe par défaut par des annotations, comme pour les routes du contrôleur.

```
/**
 * @ORM\Entity
 */
class MyEntity {
```

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

 Atelier : création d'une entité Article

L'entité comprendra pour l'instant un titre, une date de création, une date de dernière mise à jour, un corps de texte et un état (publié ou non).

## Migration de base de données

Les migrations sont un moyen efficace de garantir une intégrité des données en base. Ce sont des classes qui prévoient les transformations de la base vers sa nouvelle forme et retour. On garde un historique complet et réversible des modifications de la base de données.

Le maker propose un modèle de migration :

```
bin/console make:migration
```

On peut ensuite indiquer à Doctrine d'appliquer la ou les migrations ajoutées :

```
bin/console doctrine:migrations:migrate
```

 Atelier : migration de la nouvelle entité Article

## Fixtures

Une fixture, c'est une donnée de test générée par du code. L'idée c'est de pouvoir entrer 20 articles sans avoir à le faire à la main à chaque fois, par exemple.


On peut se faire aider par le maker comme d'habitude :

```
bin/console make:fixtures
```

Pour compléter les fixtures, il va falloir utiliser **ObjectManager** qui est un gestionnaire d'entités. Il permet de les persister et de mettre à jour les infos de la base au travers des fonctions **persist(\$entity)** et **flush()**.

On peut ensuite les charger avec une commande doctrine (qui vide la base au passage) :

```
bin/console doctrine:fixtures:load
```

 Atelier : chargement de 3 fixtures pour Article

# CRUD : créer une entité

Il existe une commande `make:crud` dans le maker mais il se passe beaucoup de choses dedans qu'il est préférable de voir en détail. Dans cette partie on se concentrera sur le fait d'ajouter une entité en base.

## Introduction aux formulaires

Pour créer des formulaires dans un contrôleur, on se sert d'un *builder*: le `FormBuilder`. Il va nous permettre de le lier à une entité si on le souhaite, d'ajouter des champs et d'en extraire un formulaire finalisé.

Un exemple au sein d'un contrôleur :

```
public function myAction()
{
    $form = $this->createFormBuilder(new MyEntity())
        ->add('myField', TextType::class)
        ->getForm();
}
```

 Atelier : création de formulaire pour Article

Vous aurez besoin de types de formulaires (dans l'exemple, `TextType`), se référer à la documentation des `FormType` Symfony.

## Création d'un `FormType`

Le but d'un contrôleur est de déléguer le plus possible à des services et de juste coordonner le tout. Il est conseillé d'extraire les formulaires dans des `FormType`, classes dédiées qui contiennent une définition de formulaire.

Le maker peut une nouvelle fois nous aider :

```
bin/console make:form
```

On peut ensuite créer une instance du formulaire dans le contrôleur :

```
public function myAction()  
{  
    $form = $this->createForm(MyEntityType::class, new  
MyEntity());  
}
```

 Atelier : extraction du formulaire dans ArticleType

Le maker ne peut pas tout deviner, il faut finir la configuration de cette classe à la main.

## Affichage de formulaire

Nous avons vu pour l'instant 3 classes qui concernent les formulaires :

- **FormBuilder**, avec lequel on crée un formulaire
- **FormType**, une définition de formulaire qui contient un build
- **Form**, le formulaire finalisé avec lequel on peut valider une requête et récupérer des données

Nous allons maintenant nous intéresser à **FormView**, une vue du formulaire que l'on peut afficher. On l'appelle à partir du formulaire :

```
$form->createView();
```

On passe le résultat à un template qui pourra afficher la vue de formulaire grâce à une fonction Twig :

```
{{ form(monFormulaire) }}
```

 Atelier : création de page nouvel article et affichage de formulaire


Ne pas oublier de définir une route, et pourquoi pas créer un lien de la page d'accueil vers le nouvel article.

## Amélioration de l'affichage

Comme on peut le voir le formulaire se retrouve sans bouton d'envoi. C'est une mauvaise pratique de l'inclure dans la définition de formulaire : un même formulaire peut servir à créer, éditer, cloner une entité ...

Une solution est d'utiliser des fonctions Twig plus fines dans l'affichage :

```
{{ form_start(my_form) }}
    {{ form_widget(my_form) }}
    <input type="submit" value="Enregistrer" />
{{ form_end(my_form) }}
```

 Atelier : adaptation de l'affichage du formulaire

## Récupération des données

On peut maintenant envoyer un formulaire mais ... rien ne se passe. Il nous faut encore récupérer les données à partir de la requête actuelle, les valider et les sauvegarder.

Pour récupérer la requête nous allons utiliser la technique du typehinting (indice de type) que l'on reverra plus tard : on ajoute un argument à notre action. Pas besoin d'appeler la fonction en lui passant effectivement cet argument : c'est Symfony qui va le chercher tout seul.

```
use Symfony\Component\HttpFoundation\Request;

public function myAction(Request $request) {}
```

Il s'agit ensuite de récupérer les données du formulaire. Si c'est la première fois qu'on arrive sur la page, les données seront absentes et le formulaire restera vide ; sinon le formulaire contiendra les données entrées.

```
$form->handleRequest($request);
```

## Validation et sauvegarde de données

Pour sauvegarder les données, il est obligatoire de les vérifier :

```
if ($form->isSubmitted() && $form->isValid()) {}
```

Les données (en l'occurrence une entité) sont accessibles par la fonction `$form->getData()`. Pour les sauvegarder, nous allons avoir besoin d'un service.

Pour déléguer le plus possible, un contrôleur a besoin d'une façon de récupérer d'autres éléments de logique métier. Dans Symfony ces éléments indépendants s'appellent *services*. Les services sont des singletons. Comme les requêtes, on peut les instancier dans un contrôleur grâce au typehinting.

Nous allons nous intéresser au service `EntityManager`, un gestionnaire d'entité de Doctrine qui étend `ObjectManager`, vu dans les fixtures, qui permet d'enregistrer les nouvelles entités et de les mettre à jour.

Quand on veut trouver un service, le plus simple est d'utiliser la console Symfony :

```
bin/console debug:container entitymanager
```

Un exemple avec `EntityManager` en action :

```
use Doctrine\ORM\EntityManager;

public function myAction(EntityManager $entityManager)
{
    // ...

    $entityManager->persist($form->getData());
    $entityManager->flush();
}
```

 Atelier : récupération et sauvegarde des données du formulaire Article

Relire les deux points précédents. L'ordre dans lequel on récupère, vérifie, sauvegarde est important.

# CRUD : lire, éditer, supprimer

## Requêter des entités

Parmi les fichiers qui ont été créés autour de notre entité, il y a ce qui s'appelle un dépôt (**ArticleRepository**). Le dépôt est un service capable de requêter des entités dans la base.

Le dépôt et le gestionnaire d'entités ont chacun leur rôle : on ne se sert pas du même service pour lire et écrire les entités.

Les dépôts qui étendent **ServiceEntityRepository** ont accès à des méthodes génériques :

- **find(\$id)** qui recherche une entité à partir de sa clé primaire
- **findAll()** qui récupère le tableau de toutes les entités
- **findBy(\$criteria)** qui permet une recherche plus précise
- etc.

🔧 Atelier : affichage des 10 articles les plus récents sur la page d'accueil

On peut également faire en sorte que seuls les articles publiés soient requêtés.

## Créer une requête personnalisée

Dans le souci habituel de ne pas surcharger le contrôleur, on peut déplacer les détails de la requête dans une méthode du dépôt.

Là, deux possibilités s'offrent à nous : utiliser les méthodes vues précédemment, ou opter pour le **QueryBuilder** qui permet de construire des requêtes plus finement. Une documentation est disponible sur [doctrine-project.org](https://doctrine-project.org).

Ça se passe de façon similaire au **FormBuilder**, mais dans le dépôt :

```
public function findSample()
```



```
{  
    $queryBuilder = $this->createQueryBuilder('name');  
}
```

 Atelier : déplacement du code dans une méthode du dépôt

On peut l'appeler `findLatest()` pour que ça soit clair.

## Édition d'une entité

Pour créer l'action, il va falloir lui passer un paramètre qui identifie l'entité clairement. Il va donc falloir créer une route avec une *wildcard*.

La syntaxe est simple :

```
/**  
 * @Route("/ma/route/{param}")  
 */  
public function myAction($param) {}
```

Le paramètre en question est complété à partir de l'url. Dans le cas où on veut récupérer une entité, on peut aller plus loin :

```
/**  
 * @Route("/ma/route/{id}")  
 */  
public function myAction(MyEntity $entity) {}
```

Symfony va de lui-même appeler le dépôt correspondant et trouver la bonne entité (ou déclencher une 404).

 Atelier : création de l'action d'édition d'un article

Très similaire à la création d'un article, sauf qu'on commence avec une entité existante au lieu d'une nouvelle.

## Suppression d'une entité

Contrairement aux actions précédentes, il est déconseillé de permettre une suppression simplement en arrivant sur une url. On va vouloir rajouter une contrainte de *méthode HTTP*. Les méthodes les plus connues sont les suivantes :

- **GET** : la méthode par défaut quand on accède à une page, elle récupère des ressources mais ne modifie rien
- **POST** : prévue pour de l'insertion de ressource, c'est la méthode par défaut pour l'envoi des formulaires Symfony
- **PUT** : méthode prévue pour une insertion partielle ou une modification de ressource
- **DELETE** : méthode prévue pour la suppression d'une ressource

Pour les utiliser, il faut passer par un formulaire, car un simple lien `<a href="#"></a>` ne peut envoyer une requête qu'avec la méthode **GET**.

On peut changer la méthode et l'action d'un formulaire lors de l'instanciation :

```
$form = $this->createForm(MyEntityType::class, null, [
    'action' => $this->generateUrl('my_route'),
    'method' => 'PUT',
]);
```

Plus de précisions sur la [documentation Symfony](#).

Pour effectivement supprimer une entité, on utilise la méthode **remove()** du service **EntityManager**.

🔧 Atelier : création d'un formulaire de suppression

🔧 Atelier bonus : routage sémantique

Si on a de l'avance, pour plus de sécurité et de cohérence, on peut adapter ses routes en suivant ce schéma :

GET	/	# page d'accueil
GET	/article	# liste de tous les articles

```
GET    /article/{id}      # vue d'un article
GET    /admin            # panneau d'administration
GET    /admin/article   # formulaire de création
d'article
POST   /admin/article   # enregistrement d'un
article
GET    /admin/article/{id} # formulaire d'édition d'un
article
PUT    /admin/article/{id} # mise à jour d'un article
DELETE /admin/article/{id} # suppression d'un article
```


# Relations entre entités

## Configuration d'une relation Doctrine

Pour Doctrine, lier ensemble des entités signifie des champs annotés à rajouter dans les entités. Quatre annotations à retenir :

- **ManyToOne** : le côté propriétaire d'une relation unique d'un seul côté
- **OneToMany** : le côté inverse de la même relation, facultatif
- **OneToOne** : une relation unique des deux côtés
- **ManyToMany** : relation multiple dans les deux sens, décrit en base par une table de jointure

On peut entrer ces annotations à la main, et pour le faire finement c'est sans doute une meilleure idée. Pour commencer cependant on peut utiliser le maker en choisissant un type de champ **relation** dans le **make:entity**.

 Atelier : création d'une entité Author liée à Article

Chaque auteur peut avoir plusieurs articles, mais les articles n'ont qu'un auteur. Les champs de l'auteur : nom à afficher, qualification (éditeur, journaliste ou freelance), date de naissance. Ne pas oublier de migrer.

À la fin de l'exercice on peut ajouter :

- un CRUD (peut-être avec la commande **make:crud** pour voir comment ça marche)
- un champ auteur dans le formulaire d'article

## Fixtures et liens entre entités

Les fixtures en particulier ont besoin d'information en plus pour pouvoir lier deux entités. Il va falloir partager les entrées créées entre différentes fixtures et les charger dans l'ordre des dépendances.

Pour partager les entrées, on va utiliser les méthodes `addReference()` et `getReference()` qui permettent d'enregistrer les objets temporairement en les référençant avec un string.

Pour les faire se charger dans le bon ordre, on va rajouter du code (à la main pour une fois). Exemple avec `ProductFixtures` qui dépend de `CategoryFixtures` :

```
use Doctrine\Common\DataFixtures
\DependentFixtureInterface;

class ProductFixtures extends Fixture implements
DependentFixtureInterface
{
    // ...

    public function getDependencies()
    {
        return [CategoryFixtures::class];
    }
}
```

 Atelier : création de `AuthorFixtures` et attribution d'auteurs pour `ArticleFixtures`

# Gestion des messages

## Locales

Dans une application on évite de laisser du texte en dur dans un template ou un contrôleur. Pour ça il y a les *messages*, rangés dans le dossier `translations/` au format `domaine.langue.format`.

Le format le plus simple à utiliser est le YAML, et le domaine par défaut est `messages`.

Pour traduire simplement dans un template Twig, il suffit d'utiliser le filtre `{{ 'message_id' | trans }}`, d'avoir écrit le message en question et d'avoir configuré la locale par défaut.

L'équivalent dans un contrôleur est la fonction `$this->trans()` qui prend en premier paramètre l'identifiant du message.

Un exemple :

```
home:
  title: Bonjour tout le monde
  intro: |
    Bienvenue ici !

    C'est chouette de vous voir ;)
```

L'identifiant de l'intro de page d'accueil est `home.intro`.

 Atelier : extraction des messages des templates et contrôleurs


## Localisation des formulaires

Par défaut, les labels des champs de formulaire sont « humanisés » sans être traduits. Pour les traduire, le plus simple est d'utiliser une option appelée `label_format`.

Exemple dans un `FormType` :

```
public function configureOptions(OptionsResolver
$resolver)
{
    $resolver->setDefaults([
        // ...
        'label_format' => 'my_entity.%name%',
    ]);
}
```

Cette option va déclencher l'utilisation du traducteur, comme la fonction `trans()`.

 Atelier : extraction des champs de l'entité Article

## Messages flash

Parfois on a besoin d'une notification qui n'apparaîtra qu'une fois. C'est à ça que servent les messages flash.

On peut en ajouter depuis le contrôleur :

```
public function monAction()
{
    $this->addFlash('success', 'mon message');
}
```

On peut les afficher dans les templates, c'est une liste à deux niveau disponible dans l'environnement de Twig :

```
{% for category, messages in app.flashes %}
{% for message in messages %}
<p class="alert alert--{{ category }}">{{ message
}}</p>
{% endfor %}
{% endfor %}
```

 Atelier : ajout de messages de succès et d'erreur aux actions des formulaires

# Tests fonctionnels et unitaires

Pour les tests avec Symfony on s'appuie sur PHPUnit, installé en dépendance composer avec le squelette de site web.

Le but est d'éviter autant que possible les régressions, courantes sur les projets importants.

## Tests unitaires

On crée une classe de test unitaire avec `make:unit-test`. On peut ensuite tester des cas spécifiques en utilisant des *assertions*. Il y en a toute une série disponible sur la documentation de [phpunit.de](http://phpunit.de).

Exemple :

```
class MyTest extends TestCase
{
    public function testAdd()
    {
        $a = 1;
        $b = 5;
        $c = 6;

        $this->assertEquals($a + $b, $c);
    }
}
```

On peut ensuite lancer le test avec `bin/phpunit`.

 Atelier : test de création d'un article et vérification des valeurs assignées

## Data providers et dépendances

PHPUnit a aussi un set d'annotations qui permet d'automatiser et de simplifier certains tests. On va se limiter à la présentation de deux annotations : `@dataProvider` et `@depends`. Il en existe beaucoup d'autres.



`@dataProvider` permet de donner à notre test plusieurs jeux de données à tester et de le faire tourner une fois pour chaque jeu :

```
/**
 * @dataProvider somethingProvider
 */
public function testSomething($a, $b, $c)
{
    $this->assertEquals($a * $b, $c);
}

public function somethingProvider()
{
    return [
        [5, 5, 25],
        [0, 5, 0],
        [7, 3, 21],
    ];
}
```

`@depends` permet de chaîner deux tests en passant des données de l'un à l'autre :

```
public function testCreation(): array
{
    $myArray = [1, 2, 3];

    $this->assertCount(3, $myArray);

    return $myArray
}

/**
 * @depends testCreation
 */
public function testIndex($myArray)
{
    $this->assertEquals($myArray[2], 3);
}
```

 Atelier : 2e test de l'article qui dépend de sa création

## Création de doublures

En anglais *mock* ou *test doubles*, il s'agit d'objets périphériques à la classe que l'on choisit de tester, et dont on va simuler un comportement pour qu'il soient aussi prévisibles que possible. On utilise la méthode `createMock()`.

Par exemple, si je teste un produit et que je veux lui attacher une catégorie, plutôt que de créer une vraie catégorie qui peut amener de nouveaux bugs, je peux créer une doublure :

```
public function testCategoryAddition()
{
    // ...

    $category = $this->createMock(Category::class);

    $category->method('getId')->willReturn(3);
    $category->method('getName')->willReturn('Sports');

    $this->assertSame('Sports', $category->getName());
}
```

 Atelier : test d'un ajout d'auteur à l'article

## Tests fonctionnels

Le but est de simuler un accès navigateur pour tester que l'application fonctionne (contrairement aux tests unitaires qui testent une classe).

On va se servir de deux outils pour ça : le *client* et le *crawler*. Avec le client, on navigue de page en page ; avec le crawler, on navigue dans une page. Chaque requête au client qui renvoie une page renvoie en fait un crawler.

La commande pour créer le test fonctionnel est `make:functional-test`.

Exemple :

```
class MyTest extends WebTestCase
{
```

```
public function testHome()  
{  
    $client = static::createClient();  
  
    $crawler = $client->request('GET', '/home');  
  
    $this->assertEquals(  
        200,  
        $client->getResponse()->getStatusCode()  
    );  
}  
}
```

 Atelier : test de l'accès à la page d'accueil

# Sécurité

## Création d'utilisateur

On peut utiliser le maker à nouveau :

```
bin/console make:user
```

Puis migrer notre entité `User` :

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

À partir de là on peut créer des fixtures d'utilisateur en se servant du service `UserPasswordEncoder` pour générer un mot de passe.

🔧 Atelier : création de 2 utilisateurs et d'une page pour les visualiser

## Formulaire de login

À nouveau, on va passer par le maker, surtout que cette génération est assez complexe :

```
bin/console make:auth
```

La nouveauté, à part un template et un contrôleur, c'est le *guard authenticator*. Il a été ajouté à la configuration de sécurité (`config/packages/security.yml`) et va donc vérifier tout seul la validité d'une tentative de login.

C'est une masse de code assez complexe à laquelle on va juste faire confiance pour l'instant. Par contre, il y a une chose à compléter dedans : la redirection en cas de succès de l'authentification. Le code entré par défaut ne risque pas de fonctionner :

```
// For example : return new  
RedirectResponse($this->router->generate('some_route'));  
throw new \Exception('TODO: provide a valid redirect  
inside '.__FILE__);
```

## Atelier : création d'un formulaire de login fonctionnel

On peut en profiter pour customiser le template de login en mettant les messages dans les traductions.

### Restriction d'accès

Les restrictions / autorisations sont la plupart du temps appliquées à des rôles. Par défaut, chaque utilisateur a au moins le rôle `ROLE_USER`. Il y a aussi 3 rôles spéciaux :

- `IS_AUTHENTICATED_ANONYMOUSLY` : utilisateur anonyme mais suivi
- `IS_AUTHENTICATED_REMEMBERED` : s'est connecté, mais potentiellement par l'intermédiaire d'un cookie
- `IS_AUTHENTICATED_FULLY` : s'est connecté par un formulaire de login

C'est une hiérarchie, donc quand on est *fully* on est aussi *remembered* et *anonymously*.

On peut restreindre l'accès à certaines url par le fichier de configuration de sécurité, par une annotation, ou par du code.

Dans `config/packages/security.yml` :

```
security:
  access_control:
    - { path: ^/admin, roles: ROLE_ADMIN }
```

Dans un contrôleur :

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration
\IsGranted;

/**
 * @Route("/admin")
 * @IsGranted("ROLE_ADMIN")
 */
class AdminController extends AbstractController
{}
```

Il est possible d'ajouter l'annotation sur une action seulement.


 Atelier : interdiction de modification aux utilisateurs anonymes

## Restriction de template

On peut également adapter la vue à ce à quoi l'utilisateur a accès. Si les changements sont trop importants, il est recommandé de créer un contrôleur et des templates séparés. En revanche, s'il s'agit juste de montrer / cacher des éléments, on peut se baser sur une fonction Twig :

```
{% if is_granted('ROLE_USER') %}Vous êtes connecté-e  
!{% endif %}
```

On peut se servir des utilisateurs pour afficher des informations sur la personne connectée. On accède à l'utilisateur connecté dans le contrôleur grâce à `$this->getUser()` ou directement dans le template par `app.user`.

 Atelier : retrait des liens de modification aux utilisateurs anonymes et affichage de l'utilisateur

## Déconnexion

Il faut encore un peu de configuration pour mettre en place une déconnexion. On indique une route dans la configuration de sécurité :

```
security:  
  firewalls:  
    main:  
      logout:  
        path: logout_route
```

Puis on crée cette route directement dans la configuration (`config/routes.yml`), pas besoin de contrôleur :

```
logout_route:
```

path: /logout

## Voters

On peut gérer les accès encore plus finement avec les *voters*, un système qui permet de définir ses propres règles pour une autorisation. Exemple : un employé est autorisé à changer son mot de passe, mais pas celui de quelqu'un d'autre.

Les voters seront utilisés avec une version « code » de la configuration d'accès : la méthode de contrôleur `denyAccessUnlessGranted()`. Cette méthode prend un type d'accès sous forme de string et le sujet du vote.

Exemple :

```
$this->denyAccessUnlessGranted('change_password',  
$employee);
```

La fonction Twig `is_granted()` marche de la même manière avec deux arguments.

On peut créer un voter avec la commande du maker `make:voter`. Il y a deux méthodes à configurer :

- `supports()` : renvoie un booléen qui indique si le voter est capable de voter sur le type d'accès
- `voteOnAttribute()` : renvoie un booléen qui donne l'accès ou non selon le sujet

Enfin, il y a plusieurs stratégies de décisions qui impliquent tous les voters en même temps. Si plusieurs voters décident qu'ils sont compétents à décider sur un type d'accès, alors selon la stratégie :

- `affirmative` : stratégie par défaut, un voter suffit à garantir l'accès
- `consensus` : l'accès est fourni s'il y a plus de oui que de non
- `unanimous` : l'accès est fourni seulement si tous les voters le valident

 Atelier : limiter le droit de suppression à l'auteur d'un article

Il va falloir modifier l'auteur pour le lier à un utilisateur, migrer, et modifier les fixtures pour qu'elles soient valides.



# Déploiement

Le déploiement est de préférence laissé aux administrateurs système et autres devops, mais il y a des éléments spécifiques à Symfony qu'il est bon de maîtriser.

## Installation sur le serveur de prod

Comme pour une installation, il faut les pré-requis de PHP et Composer. Certains hébergements simplifient la tâche, autrement il s'agira de cloner le projet puis de l'installer avec un `composer install --no-dev`.

Le principal élément à configurer est l'environnement, qui peut rester dans un `.env.local` ou passer en variables sur le serveur. Notamment, la variable `APP_ENV` doit passer en `prod` pour éviter de donner accès au mode développement.

Il faudra ensuite migrer le schéma puis les données si des données réelles sont à transférer.

Enfin, on nettoie le cache :

```
APP_ENV=prod APP_DEBUG=0 php bin/console cache:clear
```

# Bundles populaires

## Webpack Encore

*Webpack* est un outil de gestion d'assets (js, css, images, etc). Il permet de transpiler, minifier, assembler des assets depuis de nombreux langages.

*Webpack Encore* est le bundle qui lie Webpack et Symfony au travers d'une simple configuration en Node.js.

## Easy Admin Bundle

Met en place un back office complet basé sur des entités et une configuration simple en YAML.

## FOS REST Bundle

Pour gérer une API REST au travers d'une configuration.

## Hautelook Alice Bundle

Outil basé sur *Faker* pour aider à la génération de fixtures à partir d'une configuration en YAML.

## Vich Uploader

Se base sur des annotations à rajouter sur les entités pour faciliter l'upload de fichiers.

## Doctrine Extensions

Des annotations pour l'ORM Doctrine qui permettent d'ajouter aux entités : slugs, timestamps, traductions de contenus, logs de modifications ...

## FOS User Bundle

Plus très utile pour la version 4.2 de Symfony, permettait de simplifier une gestion

complexe d'utilisateurs et de droits.

# Différences avec Symfony 3

Symfony 3.4 est une version *LTS* (maintenue jusqu'à fin 2020), il peut arriver de tomber sur un projet écrit avec cette version du framework.

## Organisation du projet

L'organisation ancienne de Symfony est plus complexe, et pensée autour d'un bundle principal, le **AppBundle**.

```
.
├── app # configuration de l'application et ressources partagées
│   ├── config
│   └── Resources
├── bin # commandes symfony
├── src # répertoire du code métier de l'application
│   ├── AppBundle
│   ├── ...
│   └── Resources
├── tests # tests unitaires et fonctionnels
├── var # journaux, cache, fichiers temporaires
├── vendor # dépendances Composer
└── web # dossier public, point d'entrée de l'application
```

## Autres différences

- Symfony Maker n'existe pas : on utilise le **DoctrineGeneratorBundle**, qui est moins puissant
- Symfony Flex n'existe pas, la configuration supplémentaire est à entrer à la main quand on installe un bundle
- On ne peut pas *typehint* les dépôts Doctrine
- Symfony 3 a beaucoup plus de dépendances installées par défaut

- La configuration `.env` est placée dans un fichier de config `parameters.yml`
- Symfony 3 accepte PHP 5.4+
- L'installation d'une nouvelle application ne passe pas directement par Composer mais par un binaire Symfony spécifique
- ...

# Ressources

- [symfony.com](https://symfony.com), la documentation officielle de Symfony
- [symfonycasts.com](https://symfonycasts.com) anciennement Knp University, tutoriels et screencasts Symfony
- [packagist.org](https://packagist.org), le principal dépôt Composer
- [phpunit.de](https://phpunit.de), la documentation officielle de PHPUnit

*K. Thommy pour Dawan, 2018*