

# Symfony Framework

## PHP 8



ÉcoleRi7

# Maîtrisez le développement web de haut niveau avec la Formation Symfony

La formation Symfony est destinée aux développeurs qui souhaitent améliorer leurs compétences en matière de développement web. Elle vise à fournir aux participants une compréhension approfondie du framework Symfony, ainsi qu'à leur donner **les connaissances et les compétences** nécessaires pour créer des applications web de qualité supérieure.



Au cours de la formation, vous apprendrez comment utiliser les différents composants de Symfony pour construire des applications web **robustes et évolutives**.

Vous découvrirez également comment travailler avec les différents modèles de données, comment gérer les utilisateurs et les autorisations, et comment implémenter des fonctionnalités avancées telles que :

- Les formulaires
- Les validations
- Les routes
- Les contrôleurs
- Les vues
- Les services
- Les événements
- Les commandes
- Les tests
- Les migrations

## Élevez vos compétences en équipe, domptez les défis du web et brillez sur le Marché ! 😎

En outre, la formation vous permettra de développer **vos capacités à travailler en équipe, à communiquer avec d'autres développeurs** et à **gérer les défis du développement web**. Les participants auront également l'opportunité de mettre en pratique leurs compétences en travaillant sur des projets concrets.



En somme, la formation Symfony est une occasion unique pour les développeurs de développer leurs compétences en développement web et de se faire remarquer sur le marché du travail.



## Objectifs d'apprentissage

À la fin de cette session, les participants seront capables de :

- Configurer un environnement de développement Symfony complet en utilisant Docker et Gitpod/Codespace.
- Comprendre et utiliser les versions récentes de PHP et Node.js dans un contexte Symfony.
- Développer une application web en utilisant Symfony 7, intégrant une base de données PostgreSQL et une interface administrateur avec EasyAdmin et Twig.

## Outils et Technologies Utilisés

- **Gitpod / Codespace (GitHub)** : Fournit un environnement de développement intégré dans le cloud qui facilite la configuration et le partage des environnements de développement.
- **Docker** : Utilisé pour créer, déployer et gérer des applications dans des conteneurs isolés, assurant la cohérence entre les environnements de développement et de production.

## Au programme

- Nouveau projet Symfony 7
- **Base de données avec PostgreSQL** : Intégration et utilisation de PostgreSQL pour la gestion des données.
- **Interface web (EasyAdmin, Twig)** : Création d'interfaces utilisateur dynamiques et administratives.
- **Environnement de développement** : Setup et configuration pour un développement fluide.
- **Entité, Contrôleur, Event** : Définition des modèles de données, gestion des requêtes et réponse aux événements.
- **Messages** : Gestion de la communication entre les différents composants de l'application.
- **Workflows** : Implémentation de workflows Symfony pour automatiser des processus métier complexes et gérer des états d'entités.
- **Redisq** : Utilisation de Redis pour la gestion des files d'attente et des tâches asynchrones.

## Exemples Pratiques

- Des exemples de code seront fournis pour chaque module pour illustrer l'utilisation pratique des technologies et outils introduits.

# 1. Environnement de travail

## Editeur en ligne Gitpod

### Avantages de l'utilisation d'éditeurs de code en ligne pour la formation Symfony

- Accès depuis n'importe où, à tout moment
- Interface utilisateur conviviale
- Fonctionnalités de collaboration en temps réel
- Intégration transparente avec des services tels que Git
- Expérience de développement uniforme pour tous les participants
- Possibilité de se concentrer sur le développement des compétences
- Accès aux mêmes outils et fonctionnalités pour tous les participants
- Facilitation de la collaboration en équipe et du partage de connaissances

En utilisant des éditeurs de code en ligne pour la formation Symfony, vous bénéficierez d'une expérience plus souple et plus facile, tout en garantissant une expérience de développement uniforme pour tous les participants.



[Go to Gitpod](#)

## Mise en place de Gitpod

⚠️ Gitpod offre **50 heures d'utilisation gratuite** chaque mois dans sa version légère de l'éditeur de code en ligne. Vous devrez également associer votre compte LinkedIn pour bénéficier de cette offre.

- ➡️ **Créer un nouveau dépôt git**

Gitpod embarque une image par défaut accessible à partir de <https://hub.docker.com/r/gitpod/workspace-full>. Toutefois, il est possible de personnaliser votre environnement de développement en configurant, par exemple, les extensions de PHP.

Pour spécifier l'utilisation d'une image Docker à partir d'un fichier **Dockerfile**, il est nécessaire d'ajouter un fichier **.gitpod.yml** à la racine du projet git avec le contenu suivant :

```
image:  
  file: .gitpod.Dockerfile
```

- ➡️ **Ajouter le fichier `.gitpod` à votre nouveau projet git**

- ➔ Ensuite, il faut créer un fichier nommé `.gitpod.Dockerfile` à la racine du projet et y ajouter le contenu suivant :

```
FROM gitpod/workspace-full:latest
RUN sudo apt update
RUN sudo apt install -y apt-utils apt-transport-https postgresql postgresql-contrib
RUN sudo install-packages php-intl php-redis php-amqp php-pgsql
RUN curl -1sLf 'https://dl.cloudsmith.io/public/symfony/stable/setup.deb.sh' | sudo -E bash
RUN sudo apt install symfony-cli
```

Ce fichier **Dockerfile** spécifie l'utilisation de l'image de base `gitpod/workspace-full` et installe des paquets supplémentaires nécessaires, tels que `php-intl`, `php-redis` et `php-amqp`. Il installe également `apt-utils` et `apt-transport-https` pour la gestion des paquets, et installe `symfony-cli` en utilisant les scripts d'installation fournis par Symfony. Bien sûr, le contenu de ce fichier peut être modifié en fonction des besoins spécifiques de chaque projet.

Puis, committez et poussez ces deux fichiers sur votre branche principale.

- ➔ Rendez-vous sur <https://gitpod.io>
- ➔ Si c'est votre première fois, donnez les droits à gitpod d'accéder à votre repo, puis créez un workspace
- ➔ Sinon rendez-vous sur l'onglet "Project", cliquez sur "New Project".
- ➔ Choisissez le répertoire git préalablement configuré.

Assurez-vous de suivre ces étapes pour vous préparer à utiliser l'éditeur de code en ligne pour la formation Symfony.

# Verification de l'environnement Gitpod

## Préparation pour la formation

Sur l'éditeur de code en ligne, certains logiciels sont déjà installés. Vérifions que nous avons tous les outils nécessaires pour cette formation.

- ➔ Assurez-vous de vérifier les outils nécessaires avant de commencer la formation avec un éditeur de code en ligne.

software	command	version
docker	<code>docker --version</code>	+26.10
docker compose	<code>docker compose version</code>	+2.23
php	<code>php -version</code>	+8.3
composer	<code>symfony composer --version</code>	+2.7
node	<code>node --version</code>	+20
yarn	<code>yarn --version</code>	+1.22

# Initialisation du projet avec Symfony

- Nous allons initialiser le projet avec le client Symfony.
- Nous utiliserons la **version 7.0+**, la dernière en date.
- Nous utiliserons également **PHP 8.3+**.

Assurez-vous de suivre ces étapes pour initialiser correctement votre projet avec Symfony en utilisant les dernières versions disponibles.

- ➔ Lancer la commande suivante depuis le terminal pour initialiser un nouveau projet Symfony :

```
symfony new --dir=guestbook --webapp --version=7.0
rm -rf guestbook/.git
shopt -s dotglob
mv guestbook/* ./
rm -rf guestbook/
```

- **new** L'argument permet de construire un nouveau projet
- **--dir=guestbook** répertoire d'installation du projet symfony
- **--webapp** L'option installe tous les paquets dont vous avez généralement besoin pour créer des applications Web.

Le binaire symfony fournit également un outil permettant de vérifier si votre ordinateur répond à toutes les exigences. Ouvrez votre terminal de console et exécutez cette commande :

```
symfony book:check-requirements
```

## IDE et Extensions

Vous pouvez utiliser n'importe quel éditeur de code pour travailler sur votre projet Symfony. Cependant, pour une expérience de développement optimale, nous vous recommandons d'utiliser Visual Studio Code (VSCode) avec les extensions suivantes :

- PHP Debug
- PHP Intelephense
- PHP Namespace Resolver
- DotENV
- YAML
- PHP 8 Getter & Setter
- Extension Twig Language 2

# Rappel sur quelques notions de php orienté objet

- **Les classes** : Les classes sont des modèles pour les objets. Elles définissent les propriétés et les méthodes des objets.

```
class ClassName { /** propriétés, méthodes, ... */}
```

- **Le constructeur** : Le constructeur est une méthode spéciale qui est appelée lorsqu'un objet est instancié.

```
public function __construct() { /** code */}
```

- **Les objets** : Les objets sont des instances de classes. Ils ont des propriétés et des méthodes.

```
$object = new ClassName();
```

- **Les propriétés** : Les propriétés sont les variables d'un objet. Elles contiennent les données de l'objet.

```
public string $property = 'value';
```

- **Les méthodes** : Les méthodes sont les fonctions d'un objet. Elles permettent de manipuler les données de l'objet.

```
public function method() { /** code */}
```

- **La visibilité** : La visibilité est le fait de définir l'accès aux propriétés et aux méthodes d'une classe.

```
public $property; // accessible partout
protected $property; // accessible dans la classe et les classes héritées
private $property; // accessible uniquement dans la classe
```

- **Les espaces de noms** : Les espaces de noms sont des moyens de regrouper des classes, des interfaces, des fonctions et des constantes dans un espace de noms.

```
namespace NamespaceName;  
class ClassName { /** code */}  
  
new NamespaceName\ClassName();
```

- **La constante** : La constante est une valeur qui ne peut pas être modifiée une fois qu'elle a été définie.

```
public const string CONSTANT_NAME = 'value';
```

- **L'encapsulation** : L'encapsulation est le fait de regrouper les propriétés et les méthodes d'un objet dans une classe.

```
class ClassName  
{  
    private string $property;  
    public function getProperty() { return $this->property; }  
    public function setProperty($value) { $this->property = $value; }  
}
```

- **Le typage** : Le typage est le fait de définir le type des propriétés et des paramètres des méthodes.

```
private readonly string $property;  
public function method(string $param) : bool { return true; }
```

- **Les enums** : Les enums sont des listes de valeurs constantes.

```
enum EnumName : string { VALUE1 = 'value1', VALUE2 = 'value2', VALUE3 = 'value3'}  
  
private EnumName $property = EnumName::VALUE1;
```

- **L'héritage** : L'héritage est le fait de créer une nouvelle classe à partir d'une classe existante. La nouvelle classe hérite des propriétés et des méthodes de la classe existante.

```
class ChildClass extends ParentClass { /** code */}
```

- **Le polymorphisme** : Le polymorphisme est le fait de redéfinir les méthodes d'une classe dans une classe héritée.

```
class ParentClass { public function method() { /** code */ } }
class ChildClass extends ParentClass { public function method() { /** code */ } }
```

- **L'abstraction** : L'abstraction est le fait de définir des classes abstraites qui ne peuvent pas être instanciées directement.

```
abstract class ClassName { /** code */}
```

- **L'interface** : L'interface est le fait de définir un ensemble de méthodes que les classes doivent implémenter.

```
interface InterfaceName { public function method(); }
```

- **Le trait** : Le trait est le fait de regrouper des méthodes qui peuvent être réutilisées dans plusieurs classes.

```
trait TraitName { public function method() { /** code */ } }
class ClassName { use TraitName; }
```

- **Injection de dépendance** : L'injection de dépendance est le fait de passer des objets à une classe au lieu de les instancier à l'intérieur de la classe.

```
class ClassName { public function __construct(Dependency $dependency) { /** code */ } }
```

## 2. En route vers symfony 7

<https://symfony.com/doc/current/the-fast-track/fr/index.html>



# Présentation du projet

Nous devons trouver un projet sur lequel travailler. C'est un certain défi car nous devons choisir un projet assez vaste pour couvrir complètement Symfony, mais en même temps, il devrait être assez petit ; Afin que vous ne vous ennuyiez pas à implémenter des fonctionnalités similaires plusieurs fois.

## Description du projet

**Le projet a pour but d'obtenir un retour d'expérience sur les conférences :**

Une liste des conférences sur la page d'accueil ainsi qu'une page pour chacune d'entre elles, pleine de commentaires sympathiques. Un commentaire est composé d'un petit texte et d'une photo, optionnelle, prise pendant la conférence.

Le projet comprendra plusieurs applications :

- **Une application web traditionnelle avec une interface HTML**
- **une API**
- **une SPA pour les téléphones mobiles.**

## La maîtrise s'acquiert par la pratique

- La maîtrise s'acquiert par la pratique. Point final. Lire un livre sur Symfony, c'est bien.
- Coder une application sur votre ordinateur tout en lisant un livre sur Symfony, c'est encore mieux.
- Cette formation est très spécial puisque tout a été fait pour que vous puissiez suivre et coder.



# Structure

Jetons un coup d'oeil à la structure des répertoires :

```
your-project/
├── assets/
├── bin/
│   └── console
├── config/
├── public/
│   └── index.php
└── src/
    └── ...
├── templates/
├── tests/
└── translations/
var/
├── cache/
└── log/
└── ...
└── vendor/
└── .env
```

Comme Git est installé sur notre machine, `symfony new` nous a également créé un dépôt Git, dans lequel a été ajouté le tout premier commit.

- **bin/** contient le principal point d'entrée de la ligne de commande : `console`. Vous l'utiliserez tout le temps.
- **config/** est constitué d'un ensemble de fichiers de configuration sensibles, initialisés avec des valeurs par défaut. Un fichier par paquet. Vous les modifierez rarement : faire confiance aux valeurs par défaut est presque toujours une bonne idée.
- **public/** est le répertoire racine du site web, et le script `index.php` est le point d'entrée principal de toutes les ressources HTTP dynamiques.
- **src/** héberge tout le code que vous allez écrire ; c'est ici que vous passerez la plupart de votre temps. Par défaut, toutes les classes de ce répertoire utilisent le namespace PHP App. C'est votre répertoire de travail, votre code, votre logique de domaine. Symfony n'a pas grand-chose à y faire.
- **var/** contient les caches, les logs et les fichiers générés par l'application lors de son exécution. Vous pouvez le laisser tranquille. C'est le seul répertoire qui doit être en écriture en production.
- **vendor/** contient tous les paquets installés par Composer, y compris Symfony lui-même. C'est notre arme secrète pour un maximum de productivité. Ne réinventons pas la roue. Vous profiterez des bibliothèques existantes pour vous faciliter le travail. Le répertoire est géré par Composer. N'y touchez jamais.

C'est tout ce que vous avez besoin de savoir pour l'instant. 🎉

## Enregistrer notre travail

Prenons l'habitude après chaque chapitre de commiter notre travail. Cela nous permettra de revenir en arrière si nous rencontrons des problèmes.

- ➔ Commitez notre travail via :

```
git add .
git commit -m "Init structre"
```



# Resources publique

## Créer des ressources publiques

Tout ce qui se trouve dans le répertoire `public/` est accessible par un navigateur.

Par exemple, si vous déplacez votre fichier GIF animé (nommez-le `under-construction.gif`) dans un nouveau répertoire `public/images/`, il sera alors disponible à une URL comme <https://localhost/images/under-construction.gif>.



- ▶ Téléchargez l'image GIF animée dans le répertoire `public/images/`

```
mkdir public/images/
php -r "copy('http://clipartmag.com/images/website-under-construction-image-6.gif', 'public/images/under-construction.gif');"
```

# Lancer un serveur web local

La commande `symfony` inclut un serveur web optimisé pour le développement. Comme vous vous en doutez, il marche très bien avec Symfony. Cependant, **ne l'utilisez jamais en production.**

- ➡ À partir du répertoire du projet, démarrez le serveur web en arrière-plan (option `-d`)

```
symfony server:start -d --no-tls
```

Depuis l'éditeur en ligne [Gitpod](#) vous pouvez retrouver le lien de notre serveur lancé sur l'onglet **PORTS**. Choisissez le port **8000**, un nouvel onglet s'ouvre affichant une page "[welcome to symfony](#)"



💡 Ce n'est qu'une page de remplissage, car nous n'avons toujours pas défini de page d'accueil. Même si la page par défaut qui vous accueille est belle, c'est une page d'erreur **404**.

- ➡ Naviguez vers [/images/under-construction.gif](#). Pour percevoir notre image animé sur notre projet symfony.

*Le répertoire `public/` est le répertoire racine du site web, et le script `index.php` est le point d'entrée principal de toutes les ressources HTTP dynamiques.*

- ➡ 🏙 Commitez notre travail

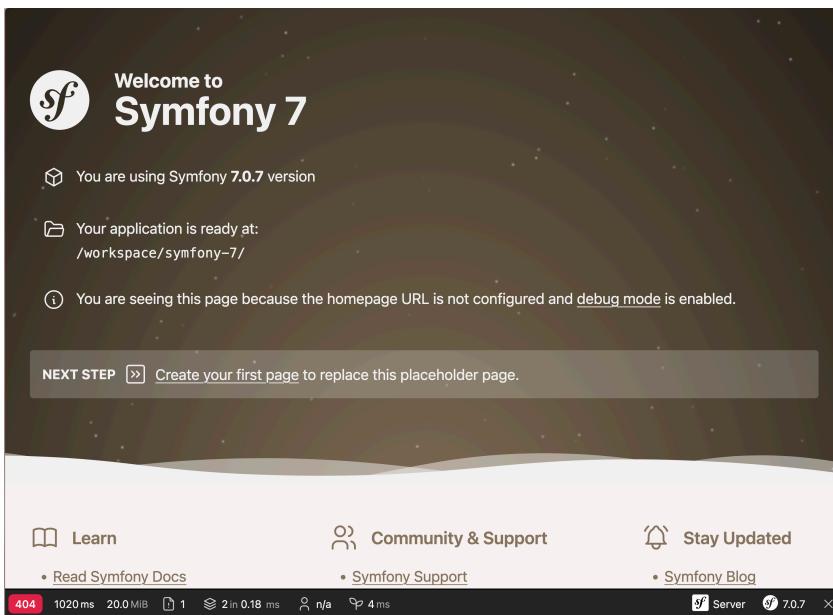
# Résolution des problèmes

Mettre en place un projet, c'est aussi avoir les bons outils pour déboguer les problèmes. Fort heureusement, des assistants très utiles sont inclus avec le paquet `webapp`.

## Découvrir les outils de débogage de Symfony

Pour commencer, Symfony Profiler vous fait gagner du temps lorsque vous devez trouver la cause première d'un problème.

Si vous regardez la page d'accueil, vous devriez voir une barre d'outils en bas de l'écran :



En utilisant Gitpod, la barre d'outils de débogage n'apparaît pas car le projet fonctionne sans `HTTPS`, alors que l'accès à Gitpod lui-même est sécurisé par `HTTPS`. Pour résoudre ce problème, il faut insérer la fonction header suivante dans le fichier `index.php` :

```
header('Content-Security-Policy: upgrade-insecure-requests');
```

# Symfony Profiler

La première chose que vous remarquerez peut-être est la **404 en rouge**. N'oubliez pas que cette page est un espace réservé car nous n'avons pas encore défini de page d'accueil. Même si la page par défaut qui vous accueille est belle, cela reste une page d'erreur. Le code d'état **HTTP** correct est donc 404 et non 200. Grâce à la barre d'outils de débogage Web, vous disposez immédiatement de l'information.

Si vous cliquez sur le petit point d'exclamation, vous obtenez le "**vrai**" message d'exception dans les **journaux du profileur Symfony**. Si vous souhaitez voir la trace de la pile, cliquez sur le lien "**Exception**" dans le menu de gauche.

Chaque fois qu'il y a un problème avec votre code, vous verrez une page d'exception comme celle-ci qui vous donne tout ce dont vous avez besoin pour comprendre le problème et d'où il vient :

The screenshot shows the Symfony Profiler interface. At the top, there's a red banner indicating an **ERROR 404** was found. Below it, the URL is listed as `http://8000-formationri7-symfony-vq1tv9tqb5o.ws-eu87.gitpod.io/?vscodeBrowserReqId=1676620710794`. The main content area is titled "Exceptions" and displays a large block of raw PHP exception log text. This log text includes details about the request, the exception type (`NotFoundHttpException`), and a stack trace with file names like `Kernel\EventListener\RouterListener.php`, `Kernel\EventListener\CompilerListener.php`, and `Kernel\EventListener\DebugEventSubscriber.php`. The sidebar on the left contains links for Request / Response, Performance, Validator, Form, Logs, Exceptions (which is currently selected), Events, Routing, Cache, Translation, Security, Debug, HTTP Client, Doctrine, and Doctrine.

Les logs sont également très utiles dans les sessions de débogage. Symfony a une commande pratique pour consulter tous les logs (du serveur web, de PHP et de votre application) :

```
symfony server:log
```

# Les environnements

## Comprendre les environnements Symfony

Comme le [Symfony Profiler](#) n'est utile que pendant le développement, nous voulons éviter qu'il soit installé en production. Par défaut, Symfony ne l'installe que pour les environnements de `dev` et de `test`.

Symfony intègre une notion d'environnement. Par défaut, il y en a trois, mais vous pouvez en ajouter autant que vous le souhaitez : `dev`, `prod` et `test`. Tous les environnements partagent le même code, mais ils représentent des configurations différentes.

**Par exemple:** tous les outils de débogage sont activés en environnement de `dev`. Dans celui de `prod`, l'application est optimisée pour la performance.

- ➡ Basculer d'un environnement à l'autre peut se faire en changeant la variable d'environnement `APP_ENV`.

## Gérer la configuration des environnements

`APP_ENV` peut être défini en utilisant des variables d'environnement "réelles" depuis votre terminal : `export APP_ENV=dev`

- ➡ Essayez de modifier la valeur de la variable `APP_ENV` à "prod", redémarrez le serveur symfony, puis rendez-vous sur la page du profiler qui n'est disponible qu'en environnement de développement.
- ➡ Pour supprimer notre variable d'environnement, vous pouvez utiliser la commande `unset APP_ENV` dans un terminal

L'utilisation de variables d'environnement réelles est la meilleure façon de définir des valeurs comme `APP_ENV` en production. Mais sur les machines de développement, avoir à définir beaucoup de variables d'environnement peut s'avérer fastidieux. Définissez-les plutôt dans un fichier `.env`.

Un fichier sensible `.env` a été généré automatiquement pour vous lorsque le projet a été créé :

```
###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=76f040716bf0a94fa2409642b1883e55
###< symfony/framework-bundle ###
```

 N'importe quel paquet peut ajouter plus de variables d'environnement à ce fichier grâce à leur [recette utilisée par Symfony Flex](#).

Le fichier `.env` est commité sur le dépôt Git et liste les valeurs par défaut de la production. Vous pouvez surcharger ces valeurs en créant un fichier `.env.local`. Ce fichier ne doit pas être commité : c'est pourquoi le fichier `.gitignore` l'ignore déjà.

 Ne stockez jamais des **données secrètes ou sensibles** dans ces fichiers. Nous verrons comment gérer ces données sensibles dans une autre étape.

### 3. Notre première route



## Les contrôleurs

La page d'accueil est une ennuyeuse page d'erreur 404. Corrigeons cela.

Lorsqu'une requête HTTP arrive au serveur, comme pour notre page d'accueil (<http://localhost:8000/>), **Symfony** essaie de trouver une route qui corresponde au chemin de la requête (🔗 ici). Une route est le lien entre le chemin de la requête et un callable PHP, une fonction devant créer la réponse HTTP associée à cette requête.

Ces callables sont nommés "**contrôleurs**". Dans Symfony, la plupart des contrôleurs sont implémentés sous la forme de classes PHP. Vous pouvez créer ces classes manuellement, mais comme nous aimons aller vite, voyons comment Symfony peut nous aider.

## Se faciliter la vie avec le Maker Bundle

Pour générer des contrôleurs facilement, nous pouvons utiliser le paquet `symfony/maker-bundle`, qui a été installé en tant que composant du paquet `webapp`.

Le **Maker Bundle** vous permet de générer un grand nombre de classes différentes. Nous l'utiliserons constamment dans cette formation. Chaque "générateur" correspond à une commande et chacune d'entre elles appartient au même namespace `make`.

La commande `list`, intégrée nativement à la console symfony, permet d'afficher toutes les commandes disponibles sous un namespace donné. Utilisez-la pour découvrir les générateurs fournis par **Maker Bundle** :

```
symfony console list make
```

# Générer un contrôleur

-  Créez votre premier Controller avec la commande `make:controller` :

```
symfony console make:controller ConferenceController
```

La commande crée une classe `ConferenceController` dans le répertoire `src/Controller/`. La classe générée contient du code standard prêt à être ajusté :

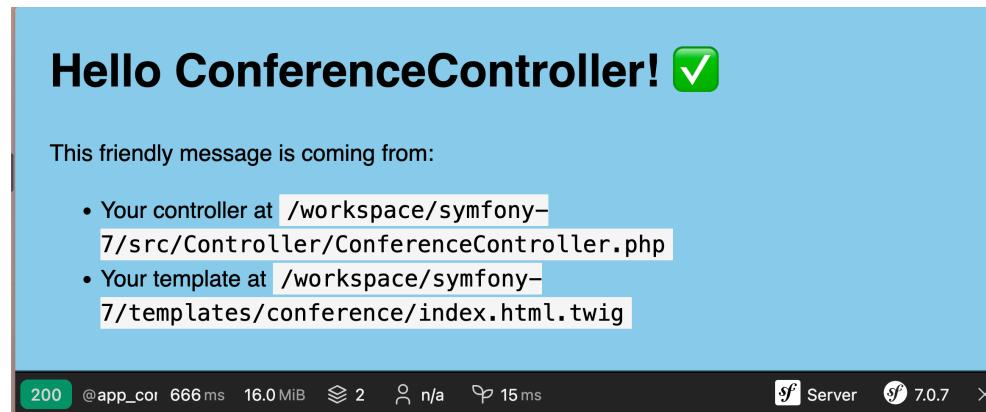
```
# src/Controller/ConferenceController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ConferenceController extends AbstractController
{
    #[Route('/conference', name: 'conference')]
    public function index(): Response
    {
        return $this->render('conference/index.html.twig', [
            'controller_name' => 'ConferenceController',
        ]);
    }
}
```

Voyons ce que notre commande make a générée pour nous visuellement sur le navigateur.

- ▶ Rendez-vous sur la page `/conference` de votre navigateur.



- ▶ 📜 Commitez notre travail

```
git add .
git commit -m "Make controller conference"
```

# Personnaliser la route

L'attribut `#[Route('/conference', name: 'conference')]` est ce qui fait de la méthode `index()` un contrôleur (la configuration est à côté du code qu'elle configure).

Lorsque vous visitez la page `/conference` dans un navigateur, le contrôleur est exécuté et une réponse est renvoyée.

- ▶ **Modifiez la route afin qu'elle corresponde à la page d'accueil (`/`)**

```
class ConferenceController extends AbstractController
{
    #[Route('/conference', name: 'app_conference')]
    #[Route('/', name: 'homepage')]
    public function index(): Response
    {
```

Le nom de la route (`name`) sera utile lorsque nous voudrons faire référence à la page d'accueil dans notre code. Au lieu de coder en dur le chemin `/`, nous utiliserons le nom de la route.

- ▶ **À la place de la page par défaut, retournons une simple page HTML**

```
public function index(): Response
{
    - return $this->render('conference/index.html.twig', [
    -     'controller_name' => 'ConferenceController',
    - ]);
    + return new Response(<<<EOF
    +     <html>
    +         <body></body>
    +     </html>
    + EOF);
}
```

- ➔ Rafraîchissez le navigateur

Nous avons maintenant une page d'accueil qui affiche une image animée. Nous avons également une route nommée `homepage` qui correspond à cette page. Nous utiliserons ce nom pour faire référence à cette page dans notre code.



💡 La responsabilité principale d'un contrôleur est de retourner une réponse HTTP (classe `Response`) pour la requête (class `Request`).

- ➔ Commitez notre travail

```
git add . && git commit -m "First controller"
```

# Ajouter un easter egg

Pour montrer comment une réponse peut tirer parti de l'information contenue dans la requête, ajoutons un petit easter egg.

- ▶ Lorsqu'une requête vers la page d'accueil sera réalisée avec un paramètre d'URL comme `?hello=Fabien`, nous ajouterons du texte pour saluer la personne

```
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
+use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ConferenceController extends AbstractController
{
    #[Route('/', name: 'homepage')]
-    public function index(): Response
+    public function index(Request $request): Response
    {
        $greet = '';
+       if ($name = $request->query->get('hello')) {
+           $greet = sprintf('<h1>Hello %s!</h1>', htmlspecialchars($name));
+       }
+
        return new Response(<<<EOF
<html>
-           <body></body>
+           <body>$greet</body>
</html>

```

# Paramètre de route dynamique

Symfony expose les données de la requête à travers un objet `Request`. Lorsque Symfony voit un argument de contrôleur avec ce typage précis, il sait automatiquement qu'il doit vous le passer. Nous pouvons l'utiliser pour récupérer le nom depuis le paramètre d'URL et ajouter un titre `<h1>`.

- ➔ Dans un navigateur, rendez-vous sur `/`, puis sur `/?hello=Fabien` pour constater la différence.
- ➔ Nous aurions également pu inclure le nom directement dans l'URL :

```
class ConferenceController extends AbstractController
{
    #[Route('/', name: 'homepage')]
    public function index(Request $request): Response
    #[Route('/hello/{name}', name: 'homepage')]
    public function index(string $name = ''): Response
    {
        $greet = '';
        if ($name = $request->query->get('hello')) {
            if ($name) {
                $greet = sprintf('<h1>Hello %s!</h1>', htmlspecialchars($name));
            }
        }
    }
}
```

La partie de la route `{name}` est un paramètre de route dynamique - il fonctionne comme un joker. Vous pouvez maintenant vous rendre sur `/hello` et sur `/hello/Fabien` dans un navigateur pour obtenir les mêmes résultats qu'auparavant. Vous pouvez récupérer la valeur du paramètre `{name}` en ajoutant un argument portant le même nom au contrôleur, donc `$name`.

- ➔ Annulez les changements que nous venons juste de faire via `git checkout .`

# Débogguer des variables

La fonction `dump()` est un utilitaire de déboggage très puissant. Elle est toujours disponible et vous permet de voir le contenu de variables complexes dans un format interactif.

- ➔ Modifiez temporairement le fichier `src/Controller/ConferenceController.php` pour afficher le contenu de l'objet Request :

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
+use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ConferenceController extends AbstractController
{
    #[Route('/', name: 'homepage')]
-    public function index(): Response
+    public function index(Request $request): Response
    {
+        dump($request);
+
        return new Response(<<<EOF
            <html>
                <body>
```

Quand vous rafraîchissez la page, une icône "cible" apparaît dans la barre de déboggage; elle vous permet d'inspecter le dump. Cliquez dessus pour accéder à une page dédiée rendant la navigation plus simple.

- ➔ Annulez les changements que nous venons juste de faire via `git checkout .`

# Travaux pratique

## Objectif :

- Créer un contrôleur qui gère une liste fictive de produits avec `/products`
- Récupérer des détails au format json (sans template twig)
- Ajouter une route pour un produit spécifique en utilisant des routes dynamiques `/product/{id}`.
- Re-factoriser le code pour utiliser une seule liste de produits.
- Commitez votre travail.

- Données fictives :

```
$products = [  
    [  
        'id' => 1,  
        'name' => 'Iphone 13',  
        'price' => 999.99,  
        'description' => 'Apple iPhone 13 Pro 256 Go Bleu Pacifique',  
        'category' => 'smartphone',  
    ],  
    [  
        'id' => 2,  
        'name' => 'Samsung Galaxy S21',  
        'price' => 899.99,  
        'description' => 'Samsung Galaxy S21 5G 128 Go Double SIM Noir Phantom',  
        'category' => 'smartphone',  
    ],  
    [  
        'id' => 3,  
        'name' => 'Huawei P40',  
        'price' => 799.99,  
        'description' => 'Huawei P40 Pro 5G 256 Go Double SIM Noir',  
        'category' => 'smartphone',  
    ],  
    [  
        'id' => 4,  
        'name' => 'MacBook Pro',  
        'price' => 1299.99,  
        'description' => 'Apple MacBook Pro 13" 256 Go SSD 8 Go RAM Intel Core i',  
        'category' => 'laptop',  
    ]]
```

# 4. Gestion des données

# Base de données PostgreSQL

Le site web du livre d'or de la conférence permet de recueillir des **commentaires** pendant les **conférences**. Nous avons besoin de stocker ces commentaires dans un stockage persistant.

Un commentaire est mieux décrit par une structure de données fixe :

- Un nom
- Un email
- Le texte du commentaire
- Et une photo facultative

Ce type de données se stocke facilement dans un moteur de base de données relationnelle traditionnel.

 **Postgresql** est le moteur de base de données que nous allons utiliser.



# Installer PostgreSQL

Sur notre machine locale, nous avons décidé d'utiliser **Docker** pour gérer nos services. Le fichier `compose.yaml` généré contient déjà **PostgreSQL** en tant que service :

Extrait du fichier `compose.yaml`

```
###> doctrine/doctrine-bundle ###
database:
    image: postgres:${POSTGRES_VERSION:-16}-alpine
    environment:
        POSTGRES_DB: ${POSTGRES_DB:-app}
        # You should definitely change the password in production
        POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-!ChangeMe!}
        POSTGRES_USER: ${POSTGRES_USER:-app}
    volumes:
        - db-data:/var/lib/postgresql/data:rw
        # You may use a bind-mounted host directory instead, so that it is harder to accidentally remove the volume and lose all your data!
        # - ./docker/db/data:/var/lib/postgresql/data:rw
###< doctrine/doctrine-bundle ###
```

Un serveur **PostgreSQL** sera alors installé et certaines variables d'environnement, qui contrôlent le nom de la base de données et ses identifiants, seront configurées. Les valeurs n'ont pas vraiment d'importance.

## Extrait du fichier `compose.override.yaml`

Le fichier `compose.override.yaml` est un fichier de configuration qui permet de surcharger les valeurs par défaut du fichier `compose.yaml`.

Nous exposons également le port PostgreSQL (`5432`) du conteneur à l'hôte local. Cela nous aidera à accéder à la base de données à partir de notre machine :

```
###> doctrine/doctrine-bundle ###
database:
  ports:
    - "5432"
+   - "5432:5432"
###< doctrine/doctrine-bundle ###
```

## Lancer notre base de données via Docker compose

- ▶ Lancez Docker Compose en arrière-plan avec l'option `-d`

```
docker compose up -d --wait
```

l'option `--wait` permet d'attendre que les conteneurs soient prêts avant de continuer.

- ▶ Attendez un peu pour laisser démarrer la base de données, puis vérifiez que tout fonctionne bien

```
docker compose ps
```

! S'il n'y a pas de conteneurs en cours d'exécution ou si la colonne `STATUS` n'indique pas `Up`, vérifiez les logs de Docker Compose :

```
docker compose logs database
```

# Accéder à la base de données

L'utilitaire en ligne de commande `psql` peut parfois s'avérer utile. 🤔 Mais vous devez vous rappeler des informations d'identification et du nom de la base de données. Encore moins évident, vous devez aussi connaître le port local sur lequel la base de données tourne sur l'hôte.

👉 Si vous utilisez `psql` avec la commande `symfony`, vous n'avez pas besoin de vous souvenir de quoi que ce soit.

La commande `symfony` détecte automatiquement les services Docker en cours d'exécution pour le projet et expose les variables d'environnement dont `psql` a besoin pour se connecter à la base de données.

- ➡ Grâce à ces conventions, accéder à la base de données avec `symfony run` est beaucoup plus facile

```
symfony run psql
```

- ➡ Ou via un `docker compose exec` sur le container `database`

```
docker compose exec database psql app app
```

## Documentation PostgreSQL

- structure : <https://www.postgresql.org/docs/current/tutorial-accessdb.html>
- table & requêtes : <https://www.postgresql.org/docs/current/tutorial-table.html>

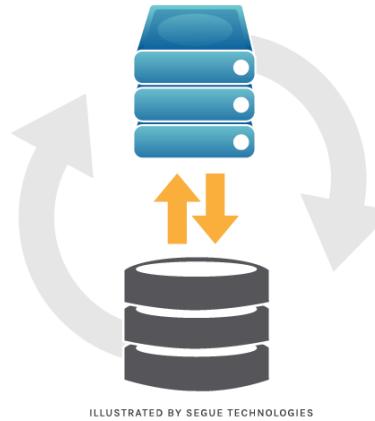
## Dump et restauration des données de la base de données

Utilisez pg\_dump pour vider les données de la base de données :

```
symfony run pg_dump --data-only > dump.sql
```

Et restaurez les données :

```
symfony run psql < dump.sql
```



## Exposer des variables d'environnement

Entre les variables d'environnement définies dans le fichier `.env` et celles définies dans le fichier `compose.yaml`, il y a beaucoup de variables d'environnement disponibles. 😊

Les mêmes nom de variables d'environnement peuvent être définis dans plusieurs fichiers, **mais les valeurs définies dans les fichiers `compose.yaml` ont la priorité** sur celles définies dans `.env`.

Pour afficher toutes les variables d'environnement exposées:

- affiche toutes les variables (server, .env, docker, ...)

```
symfony var:export --debug --multiline
```

- afficher seulement les variables dans les fichiers `.env.*`

```
symfony console debug:dotenv
```

# Configurer Doctrine ORM

Pour interagir avec la base de données depuis PHP, nous allons nous appuyer sur **Doctrine**, un ensemble de bibliothèques qui nous aide à gérer les bases de données :

- **Doctrine DBAL** (une couche d'abstraction de la base de données)
- **Doctrine ORM** (une librairie pour manipuler le contenu de notre base de données en utilisant des objets PHP)
- Et **Doctrine Migrations**.



Comment est-ce que Doctrine est au courant de notre connexion à la base de données ? 😐

👉 La recette de Doctrine (`composer recipes` au moment de l'installation du paquet) a ajouté un fichier de configuration qui contrôle son comportement : `config/packages/doctrine.yaml`.

Le paramètre principal est le `DSN` de la base de données, une chaîne contenant toutes les informations sur la connexion : identifiants, hôte, port, etc. Par défaut, Doctrine recherche une variable d'environnement `DATABASE_URL`.

🔧 Presque tous les paquets installés sont configurés dans le répertoire `config/packages/`. Les valeurs par défaut ont été choisies avec soin pour fonctionner avec la plupart des applications.

## Comprendre les conventions des variables d'environnement de Symfony

👉 Vous pouvez définir la variable `DATABASE_URL` manuellement dans le fichier `.env` ou `.env.local`.

Mais grâce à la recette du paquet, vous verrez un exemple de variable `DATABASE_URL` dans votre fichier `.env`. Mais comme le port exposé par Docker vers PostgreSQL peut changer, c'est assez lourd. Il y a une meilleure solution.

Au lieu de coder en dur la variable `DATABASE_URL` dans un fichier, nous pouvons préfixer toutes les commandes avec `symfony`. Ceci détectera les services exécutés par Docker (lorsque le tunnel est ouvert) et définira automatiquement la variable d'environnement.

💡 **Docker Compose** fonctionne parfaitement avec Symfony grâce à ces variables d'environnement.

# Configurer l'url de la base de données

- ▶ Vérifiez toutes les variables d'environnement exposées en exécutant :

```
symfony var:export --multiline | grep DATABASE_URL  
export DATABASE_URL=postgres://app:!ChangeMe!@127.0.0.1:5432/app?sslmode=disable&charset=utf8&serverVersion=16.3
```

Vous rappelez-vous du nom du service `database` utilisé dans les configurations Docker `compose.yml`? Les noms des services sont utilisés comme préfixes pour définir des variables d'environnement telles que `DATABASE_URL`. Si vos services sont nommés selon les conventions Symfony, aucune autre configuration n'est nécessaire.

- ▶ Nous allons quand même changer le fichier `.env` pour initialiser la variable `DATABASE_URL` pour l'utilisation de PostgreSQL

```
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8&charset=utf8mb4"  
-DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"  
+DATABASE_URL="postgresql://127.0.0.1:5432/db?serverVersion=15&charset=utf8"  
###< doctrine/doctrine-bundle ###
```

💡 Pourquoi l'information doit-elle être dupliquée à deux endroits différents ?

👉 Parce que sur certaines plates-formes de Cloud, au moment de la compilation, l'URL de la base de données n'est peut-être pas encore connue mais Doctrine a besoin de connaître le moteur de la base de données pour initialiser sa configuration. Ainsi, l'hôte, le pseudo et le mot de passe n'ont pas vraiment d'importance.

# Générer la classe d'entité "Conférence"

Une conférence peut être décrite en quelques propriétés :

- La ville où la conférence est organisée ;
- L'année de la conférence ;
- Une option international pour indiquer si la conférence est locale ou internationale (SymfonyLive vs SymfonyCon).

Le **Maker Bundle** peut nous aider à générer une classe (une classe `Entity`) qui représente une conférence.

- ➡ Il est maintenant temps de générer l'entité `Conference`

```
symfony console make:entity Conference
```

💡 Cette commande est interactive : elle vous guidera dans le processus d'ajout de tous les champs dont vous avez besoin.

Utilisez les réponses suivantes (la plupart d'entre elles sont les valeurs par défaut, vous pouvez donc appuyer sur la touche "Entrée" pour les utiliser) :

- `city`, `string`, `255`, `no` ;
- `year`, `string`, `4`, `no` ;
- `isInternational`, `boolean`, `no`.

La classe `Conference` a été stockée sous le namespace `App\Entity\`. La commande a également généré une classe de repository Doctrine : `App\Repository\ConferenceRepository`.

## Générer la classe d'entités "Comment"

Notez que la classe elle-même est une classe PHP sans aucune référence à Doctrine. Les attributs sont utilisés pour ajouter des métadonnées utiles à Doctrine afin de mapper la classe à sa table associée dans la base de données.

Doctrine a ajouté un attribut `id` pour stocker la clé primaire de la ligne dans la table de la base de données. Cette clé `(ORM\Id())` est générée automatiquement `(ORM\GeneratedValue())` avec une stratégie qui dépend du moteur de base de données.

- ➔ Maintenant, générez une classe d'entité pour les commentaires de la conférence :

```
symfony console make:entity Comment
```

- ➔ Entrez les réponses suivantes :

- `author`, `string`, `255`, `no` ;
- `text`, `text`, `no` ;
- `email`, `string`, `255`, `no` ;
- `createdAt`, `datetime_immutable`, `no`.

## Lier les entités

Les deux entités, `Conference` et `Comment`, devraient être liées l'une à l'autre. **Une conférence peut avoir zéro commentaire ou plus**, ce qui s'appelle une relation `one-to-many`.

- ➔ Utilisez à nouveau la commande `make:entity` pour ajouter cette relation à la classe `Conference` :

```
symfony console make:entity Conference
```

- ➔ Entrez les réponses suivantes :
- `comments`, `OneToMany`, `Comment`, `conference`, `no`, `yes`

💡 Si vous entrez `?` comme réponse pour le type, vous obtiendrez tous les types pris en charge

Les entités ont été mises à jour et sont maintenant liées l'une à l'autre.

Tout ce dont vous avez besoin pour gérer la relation a été généré pour vous. Une fois généré, le code devient le vôtre ; n'hésitez pas à le personnaliser comme vous le souhaitez.

## Ajouter d'autres propriétés

Je viens de réaliser que nous avons oublié d'ajouter une propriété sur l'entité `Comment` :

- Une photo de la conférence peut être jointe afin d'illustrer un retour d'expérience.



Allons modifier la classe `Comment` pour ajouter cette propriété.

- ➔ Exécutez à nouveau `make:entity` et ajoutez une propriété/colonne `photoFilename` de type `string`.

Mais, comme l'ajout d'une photo est facultatif, permettez-lui d'être null :

```
symfony console make:entity Comment
```

# Migrer la base de données

La structure du projet est maintenant entièrement décrite par les deux classes générées. Nous devons maintenant créer les tables de base de données liées à ces entités PHP.

**Doctrine Migrations** est la solution idéale pour cela. Le paquet a déjà été installé dans le cadre de la dépendance `orm`.

**Une migration** est une classe qui décrit les changements nécessaires pour mettre à jour un schéma de base de données, de son état actuel vers le nouveau, en fonction des attributs de l'entité. Comme la base de données est vide pour l'instant, la migration devrait consister en la création de deux tables.

- ▶ Générez la migration pour créer les tables de base de données

```
symfony console make:migration --formatted
```

Notez le nom du fichier généré, du genre `migrations/Version20191019083640.php`

## Mettre à jour la base de données locale

- ▶ Vous pouvez maintenant exécuter la migration générée pour mettre à jour le schéma de la base de données locale

```
symfony console doctrine:migrations:migrate --no-interaction
```

- ▶ 📄 Commitez notre travail

```
git add . && git commit -m "Add entities and migrations"
```

# Travaux pratique

Objectif :

- ➔ Créer une entité `Product` et `Category` en vous basant sur les données du précédent TP, pour définir les propriétés.
- ➔ Ajouter une relation `ManyToOne` entre `Product` et `Category`
- ➔ Générer la migration et mettre à jour la base de données.
- ➔ Bonus: Ajouter des données de test dans la base de données via la console `psql`

# 5. Interface d'administration

## Easy Admin

L'ajout des prochaines conférences à la base de données est le travail des admins du projet. Une interface d'administration est une section protégée du site web où les admins du projet peuvent gérer les données du site web, modérer les commentaires, et plus encore.

Comment pouvons-nous le créer aussi rapidement ? En utilisant un bundle capable de générer une interface d'administration basée sur la structure du projet. [EasyAdmin](#) convient parfaitement.



## Installer Easy Admin

Même si le package `webapp` a ajouté automatiquement de nombreux packages utiles, pour des fonctionnalités plus spécifiques, nous devons ajouter d'autres dépendances ? Avec Composer. En plus des paquets « standards » de Composer, nous travaillerons avec deux types de paquets « spéciaux » :

- *Composants Symfony*: Paquets qui implémentent les fonctionnalités de base et les abstractions de bas niveau dont la plupart des applications ont besoin (routage, console, client HTTP, mailer, cache, etc.) ;
- *Bundles Symfony*: Paquets qui ajoutent des fonctionnalités de haut niveau ou fournissent des intégrations avec des bibliothèques tierces (les bundles sont principalement créés par la communauté).
- ➔ Ajoutez `EasyAdmin` comme dépendance du projet :

```
symfony composer req "admin:^4"
```

 `admin` est un alias pour le paquet `easycorp/easyadmin-bundle`. Les alias ne sont pas une fonctionnalité interne à Composer, mais un concept fourni par Symfony pour vous faciliter la vie

# Configurer EasyAdmin

Le bundle EasyAdmin crée automatiquement une section d'administration pour votre application basée sur des contrôleurs spécifiques.

Pour débuter avec EasyAdmin

-  commençons par générer un "tableau de bord d'administration" qui sera le point d'entrée principal pour gérer les données du site.

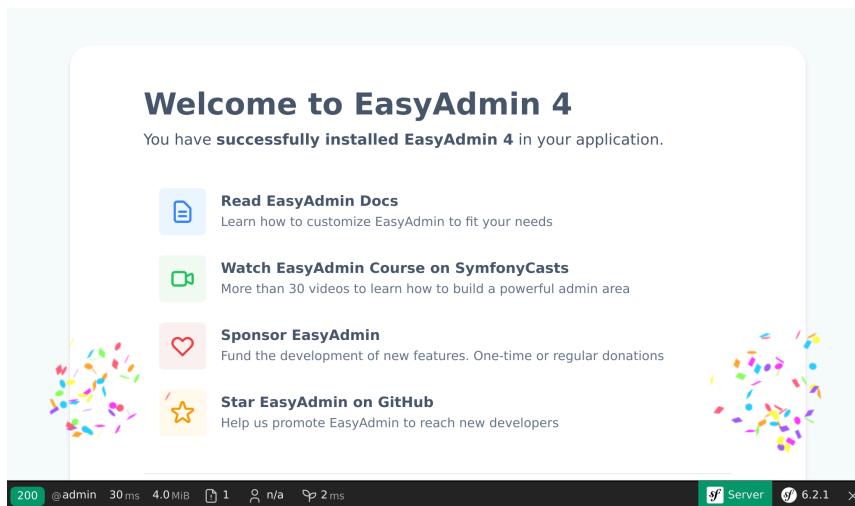
```
symfony console make:admin:dashboard
```

Avec les réponses par défaut, le maker crée le contrôleur `src/Controller/Admin/DashboardController.php`

 Par convention, les contrôleurs d'administration sont stockés dans leur propre espace de nom `App\Controller\Admin`.

# Configurer EasyAdmin

Accédez à l'interface d'administration générée grâce à l'URL `/admin` telle que configurée par la méthode `index()` (vous pouvez modifier l'URL comme bon vous semble) :



💥 Boom ! 💥

Nous avons une belle interface d'administration, prête à être adaptée à nos besoins.

## Générer un CRUD

L'étape suivante consiste à créer des contrôleurs pour gérer **les conférences** et **les commentaires**.

Dans le contrôleur du tableau de bord, vous avez peut-être remarqué la méthode `configureMenuItems()` qui contient un commentaire à propos de l'ajout de liens aux "CRUDs".

"**CRUD**" est un acronyme pour "**Create, Read, Update and Delete**", les quatre opérations de base que vous allez effectuer sur une entité. C'est exactement ce que nous voulons que notre page d'administration fasse pour nous. EasyAdmin facilite encore plus les choses en prenant en charge les fonctionnalités de filtre et de recherche.

- ➔ Générons un **CRUD** pour les conférences :

```
symfony console make:admin:crud
```

Sélectionnez `App\Entity\Conference` pour créer une interface d'administration pour les conférences et utiliser les valeurs par défaut pour les autres questions. Le fichier suivant devrait être généré : `src/Controller/Admin/ConferenceCrudController.php`

- ➔ Faites la même chose pour les commentaires :

```
symfony console make:admin:crud
```

## Lier un CRUD au Dashboard

- ▶ La dernière étape consiste à relier les CRUDs d'administration des *conférences* et des *commentaires* au tableau de bord:

```
# src/Controller/Admin/DashboardController.php
namespace App\Controller\Admin;

+use App\Entity\Comment;
+use App\Entity\Conference;
use EasyCorp\Bundle\EasyAdminBundle\Config\Dashboard;

@@ ...
public function configureMenuItems(): iterable
{
    - yield MenuItem::linkToDashboard('Dashboard', 'fa fa-home');
    - // yield MenuItem::linkToCrud('The Label', 'fas fa-list', EntityClass::class);
    + yield MenuItem::linkToRoute('Back to the website', 'fas fa-home', 'homepage');
    + yield MenuItem::linkToCrud('Conferences', 'fas fa-map-marker-alt', Conference::class);
    + yield MenuItem::linkToCrud('Comments', 'fas fa-comments', Comment::class);
}
```

Nous avons surchargé la méthode `configureMenuItems()` pour ajouter les éléments de menu avec les icônes adéquates pour les conférences et les commentaires, et pour ajouter un lien de retour vers la page d'accueil du site.

EasyAdmin expose une API pour faciliter les liaisons avec les CRUDs des entités via la méthode `MenuItem::linkToRoute()`.

## Configurer le tableau de bord

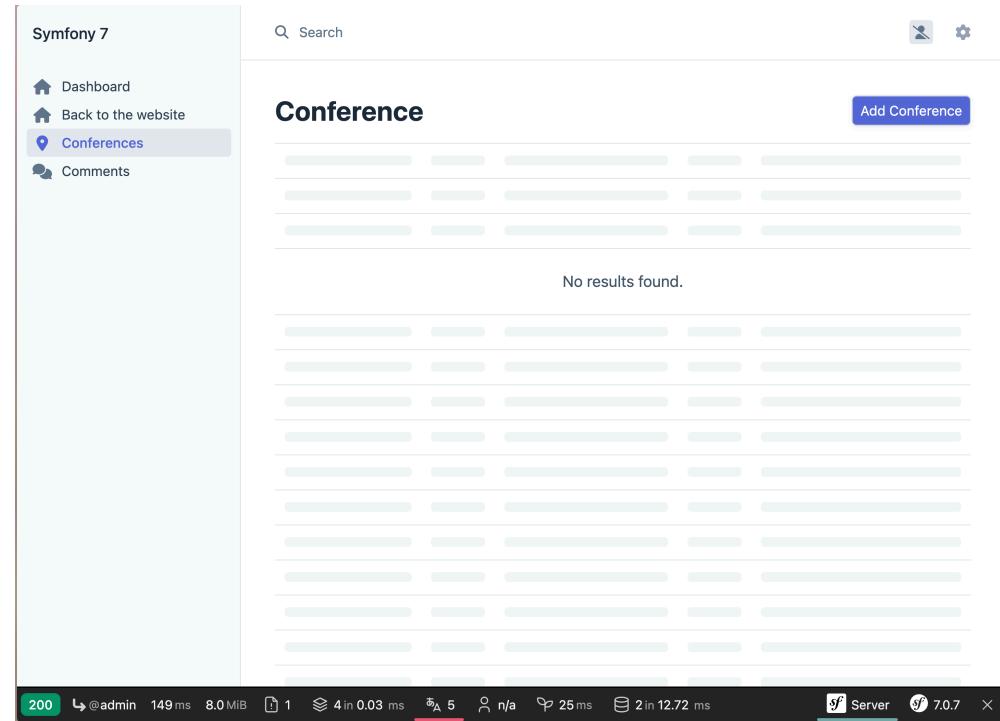
Le tableau de bord principal est vide pour le moment. C'est ici que vous pouvez afficher certaines statistiques, ou n'importe quelle information pertinente.

- ➔ Comme nous n'avons rien d'important à y afficher, redirigeons cette page vers la liste des conférences :

```
# src/Controller/Admin/DashboardController.php
use EasyCorp\Bundle\EasyAdminBundle\Controller\AbstractDashboardController;
+use EasyCorp\Bundle\EasyAdminBundle\Router\AdminUrlGenerator;
use Symfony\Component\HttpFoundation\Response;

@@ ...
public function index(): Response
{
-    return parent::index();
+    $routeBuilder = $this->container->get(AdminUrlGenerator::class);
+    $url = $routeBuilder->setController(ConferenceCrudController::class)->generateUrl();
+
+    return $this->redirect($url);
```

- ➔ Rafraîchissez la page d'administration `/admin` pour voir le résultat.



Voilà ! Nous avons maintenant une interface d'administration fonctionnelle pour gérer les conférences et les commentaires. EasyAdmin a généré pour nous des contrôleurs pour les entités `Conference` et `Comment`, et les a liés à notre tableau de bord.

# Des entités Stringable

Quand nous affichons les relations entre les entités (la conférence liée à un commentaire), EasyAdmin essaie d'utiliser la représentation textuelle de la conférence. Par défaut, il s'appuie sur une convention qui utilise le nom de l'entité et la clé primaire (par exemple `Conference #1`) si l'entité ne définit pas la méthode "magique" `__toString()`. Pour rendre l'affichage plus parlant, ajoutez cette méthode sur la classe `Conference`.

- ➔ La méthode `__toString()` fait partie du contrat de l'interface `\Stringable`. Nous devons l'implémenter pour respecter le contrat sur nos entités.

```
# src/Entity/Conference.php

- class Conference
+ class Conference implements \Stringable

@@ ....

+ public function __toString(): string
+
+ {
+     return $this->city.' '.$this->year;
+
+ }
+
+ public function getId(): ?int
```

```
# src/Entity/Comment.php

- class Comment
+ class Comment implements \Stringable

@@ ....

+ public function __toString(): string
+
+ {
+     return (string) $this->getEmail();
+
+ }
+
+ public function getId(): ?int
```

Vous pouvez maintenant **ajouter/modifier/supprimer** des conférences directement depuis l'interface d'administration.

- ➔ Jouez avec et ajoutez au moins une conférence.

Notez qu'il ne sera pas possible d'ajouter de commentaires pour l'instant. Car certains champs sont obligatoires et ne sont pas affichés dans le formulaire. Nous allons y remédier dans la prochaine étape.

# Personnaliser EasyAdmin

L'interface d'administration par défaut fonctionne bien, mais elle peut être personnalisée de plusieurs façons pour améliorer son utilisation.

-  **Faisons quelques changements simples pour montrer quelques possibilités, comme la recherche, le tri et le filtrage des données.**

```
# src/Controller/Admin/CommentCrudController.php

    return Comment::class;
}

/*
+ public function configureCrud(Crud $crud): Crud
{
    return $crud
        ->setEntityTypeInSingular('Conference Comment')
        ->setEntityTypeInPlural('Conference Comments')
        ->setSearchFields(['author', 'text', 'email'])
        ->setDefaultSort(['createdAt' => 'DESC']);
}
+
+ public function configureFilters(Filters $filters): Filters
{
    return $filters
        ->add(EntityFilter::new('conference'));
}
+
```

- ◆ La méthode `configureCrud()` permet de personnaliser le CRUD. Ici, nous avons changé le nom de l'entité, ajouté des champs de recherche, et défini un tri par défaut.
- ◆ La méthode `configureFilters()` permet de personnaliser les filtres et définissent quels filtres apparaissent au dessus du champ de recherche. Ici, nous avons ajouté un filtre pour la conférence.

# Personnaliser les champs d'un CRUD

Pour personnaliser notre CRUD

`src/Controller/Admin/CommentCrudController.php`:

- ➡️ **lister les champs de manière explicite dans la méthode `configureFields()` nous permet de les ordonner comme nous le souhaitons.**

Certains champs bénéficient d'une configuration supplémentaire, comme masquer le champ texte sur la page d'index.

```
public function configureFields(string $pageName): iterable
{
    return [
        IdField::new('id'),
        TextField::new('title'),
        TextEditorField::new('description'),
    ];
+
+    yield AssociationField::new('conference');
+    yield TextField::new('author');
+    yield EmailField::new('email');
+    yield TextareaField::new('text')
+        ->hideOnIndex();
+    yield TextField::new('photoFilename')
+        ->onlyOnIndex();

+
+    $createdAt = DateTimeField::new('createdAt')
+        ->setFormTypeOptions([
+            'html5' => true,
+            'years' => range(date('Y'), date('Y') + 5),
+            'widget' => 'single_text',
+        ]);
+
+    if (Crud::PAGE_EDIT === $pageName) {
+        yield $createdAt->setFormTypeOption('disabled', true);
+    } else {
+        yield $createdAt;
+    }
}
/*
*/
```

# Tester l'interface d'administration

Ces personnalisations ne sont qu'une petite introduction aux possibilités offertes par EasyAdmin.

- ▶ Jouez avec l'interface d'administration, filtrez les commentaires par conférence, ou recherchez des commentaires par email par exemple.

Nous avons maintenant une interface d'administration fonctionnelle pour gérer les conférences et les commentaires.

⚠ Le seul problème, c'est que n'importe qui peut accéder à cette interface.

👮 Ne vous inquiétez pas, nous la sécuriserons dans une prochaine étape.

- ▶ Committez notre travail `git add . && git commit -m "Easy admin"`

The screenshot shows a web browser displaying the 'Comments' section of the 'Conferences' entity in the EasyAdmin interface. A modal dialog box titled 'Filters' is open, with the 'Conference' checkbox selected. Inside the dialog, there is a dropdown menu set to 'is same' and a value 'Amsterdam 2019' has been selected. The main list area shows four results, each representing a comment from a different user (Thomas, Helene, Lucas) at the Amsterdam 2019 conference. The list includes columns for the conference name, author's name, email, file name, and creation date. At the bottom of the page, there is a footer with various performance metrics and a server status indicator.

Conference	Author	Email	File Name	Created At
Amsterdam 2019	Thomas	thomas@some.where	lisbon.png	Dec 7, 2022, 9:09:04 AM
Amsterdam 2019	Helene	helene@some.where	lisbon.png	Dec 7, 2022, 9:09:04 AM
Amsterdam 2019	Lucas	lucas@some.where	lisbon.png	Dec 7, 2022, 9:09:04 AM

## Travaux pratique

Objectif :

- ➔ Ajouter un CRUD pour l'entité `Product` et `Category`
- ➔ Personnaliser les champs pour les deux CRUDs
- ➔ Personnaliser les filtres pour les deux CRUDs
- ➔ Ajouter quelques produits et catégories afin de tester nos CRUDs
- ➔ Ajouter un lien depuis notre page d'accueil vers l'interface d'administration

# 6. Interface web TWIG



# Utiliser Twig pour les templates

Tout est maintenant en place pour créer la première version de l'interface du site. On ne la fera pas jolie pour le moment, seulement fonctionnelle.

Vous vous souvenez de l'échappement de caractères que nous avons dû faire dans le contrôleur, pour l'easter egg, afin d'éviter les problèmes de sécurité ? Nous n'utiliserons pas PHP pour nos templates pour cette raison. À la place, nous utiliserons **Twig**. En plus de gérer l'échappement de caractères, Twig apporte de nombreuses fonctionnalités intéressantes, comme l'héritage des modèles.

Toutes les pages du site Web suivront le même modèle de mise en page, la même structure HTML de base. Lors de l'installation de Twig, un répertoire `templates/` a été créé automatiquement, ainsi qu'un exemple de structure de base dans `base.html.twig`.

```
<!DOCTYPE html>
<html>
  <head>
    ...
    {% block stylesheets %}
    {% endblock %}

    {% block javascripts %}
      {% block importmap %}{{ importmap('app') }}{% endblock %}
    {% endblock %}
  </head>
  <body>
    {% block body %}{% endblock %}
  </body>
</html>
```

Un modèle peut définir des `blocks`. Un `block` est un emplacement où les *templates enfants*, qui étendent le modèle, ajoutent leur contenu.

# Notre premier template Twig

- ➔ Créons un template pour la page d'accueil du projet dans `templates/conference/index.html.twig`.

```
{% extends 'base.html.twig' %}

{% block title %}Conference Guestbook{% endblock %}

{% block body %}
<h2>Give your feedback!</h2>

{% for conference in conferences %}
  <h4>{{ conference }}</h4>
{% endfor %}
{% endblock %}
```

Le template étend (ou `extends`) `base.html.twig` et redéfinit les blocs `title` et `body`.

La notation `{% %}` dans un template indique des actions et des éléments de structure.

La notation `{{ }}` est utilisée pour afficher quelque chose. `{{ conference }}` affiche la représentation de la conférence (le résultat de l'appel à la méthode `toString` de l'objet `Conference`).

# Utiliser Twig dans un contrôleur

- ➔ Mettez à jour le contrôleur pour générer le contenu du template Twig

```
# src/Controller/ConferenceController.php

+use App\Repository\ConferenceRepository;
+use Twig\Environment;

class ConferenceController extends AbstractController
{
    #[Route('/', name: 'homepage')]
-    public function index(): Response
+    public function index(Environment $twig, ConferenceRepository $conferenceRepository): Response
    {
-        return new Response(<<<EOF
-            <html>
-                <body></body>
-            </html>
-        EOF
-    );
+        return new Response($twig->render('conference/index.html.twig', [
+            'conferences' => $conferenceRepository->findAll(),
+        ]));
    }
}
```

Il se passe beaucoup de choses ici.



```
public function index(Environment $twig, ...): Response
```

Pour pouvoir générer le contenu du template, nous avons besoin de l'objet `Environment` de Twig (le point d'entrée principal de Twig).

💡 Notez que nous demandons l'instance Twig en spécifiant son type dans la méthode du contrôleur. Symfony est assez intelligent pour savoir comment injecter le bon objet.

```
public function index(..., ConferenceRepository $conferenceRepository): Response
```

Nous avons également besoin du *repository* des conférences pour récupérer toutes les conférences depuis la base de données.

```
return new Response($twig->render('conference/index.html.twig', [
    'conferences' => $conferenceRepository->findAll(),
]));
```

Dans le code du contrôleur, la méthode `render()` génère le rendu du template et lui passe un tableau de variables. Nous passons la liste des objets `Conference` dans une variable `conferences`.

```
class ConferenceController extends AbstractController
```

Un contrôleur est une classe PHP standard. Nous n'avons même pas besoin d'étendre la classe `AbstractController` si nous voulons être explicites sur nos dépendances. Vous pouvez donc supprimer l'héritage (mais ne le faites pas, car nous utiliserons les raccourcis qu'il fournit dans les prochaines étapes).

# Créer la page d'une conférence

Chaque conférence devrait avoir une page dédiée à l'affichage de ses commentaires. L'ajout d'une nouvelle page consiste à ajouter un contrôleur, à définir une route et à créer le template correspondant.

- ➔ Ajoutez une méthode `show()` dans le fichier `src/Controller/ConferenceController.php`

```
+use App\Entity\Conference;
+use App\Repository\CommentRepository;
use App\Repository\ConferenceRepository;

@@ ...

+
+ #[Route('/conference/{id}', name: 'conference')]
+ public function show(Environment $twig, Conference $conference, CommentRepository $commentRepository): Response
+ {
+     return new Response($twig->render('conference/show.html.twig', [
+         'conference' => $conference,
+         'comments' => $commentRepository->findBy(['conference' => $conference], ['createdAt' => 'DESC']),
+     ]);
+ }
```

Cette méthode a un comportement particulier que nous n'avons pas encore vu. Nous demandons qu'une instance de `Conference` soit injectée dans la méthode. Mais il y en a peut-être beaucoup dans la base de données.



Symfony est capable de déterminer celle que vous voulez en se basant sur l'`{id}` passé dans le chemin de la requête (`{id}` étant la clé primaire de la table `conference` dans la base de données).

# Template de la page d'une conférence

La récupération des commentaires associés à la conférence peut se faire via la méthode `findBy()`, qui prend un critère comme premier argument.

- ➔ La dernière étape consiste à créer le fichier `templates/conference/show.html.twig`

```
{% extends 'base.html.twig' %}

{% block title %}Conference Guestbook - {{ conference }}{% endblock %}

{% block body %}
<h2>{{ conference }} Conference</h2>

{% if comments|length > 0 %}
  {% for comment in comments %}
    {% if comment.photofilename %}
      
    {% endif %}

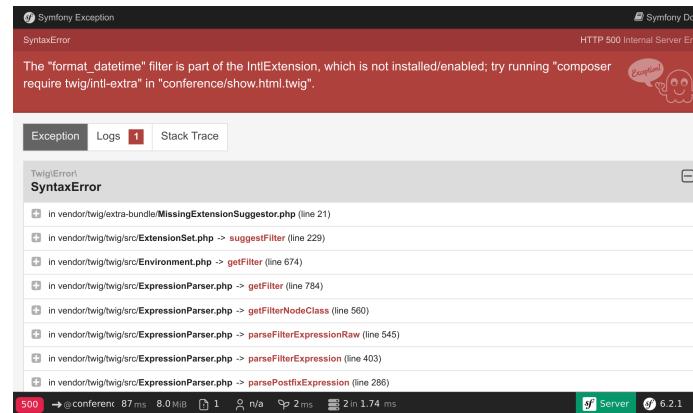
    <h4>{{ comment.author }}</h4>
    <small>{{ comment.createdAt|format_datetime('medium', 'short') }}</small>

    <p>{{ comment.text }}</p>
  {% endfor %}
  {% else %}
    <div>No comments have been posted yet for this conference.</div>
  {% endif %}
{% endblock %}
```

Dans ce template, nous utilisons le symbole `|` pour appeler les filtres Twig. Un filtre transforme une valeur. `comments|length` retourne le nombre de commentaires et `comment.createdAt|format_datetime('medium', 'short')` affiche la date dans un format lisible par l'internaute.

# Installer le composant twig intl

Essayez d'afficher la "première" conférence en naviguant vers `/conference/1`, et constatez l'erreur suivante :



L'erreur vient du filtre `format_datetime`, qui ne fait pas partie du noyau de Twig. Le message d'erreur vous donne un indice sur le paquet à installer pour résoudre le problème :

- ➔ Installez le paquet `twig/intl-extra` et rechargez la page.

```
symfony composer req "twig/intl-extra:^3"
```

Maintenant la page fonctionne correctement.

# Lier des pages entre elles

La toute dernière étape pour terminer notre première version de l'interface est:

- ➔ De rendre les pages de la conférence accessibles depuis la page d'accueil

```
# templates/conference/index.html.twig
{% for conference in conferences %}
    <h4>{{ conference }}</h4>
+   <p>
+     <a href="/conference/{{ conference.id }}">View</a>
+   </p>
{% endfor %}
{% endblock %}
```

Mais coder un chemin en dur est une mauvaise idée pour plusieurs raisons. La raison principale est que si vous transformez le chemin (de `/conference/{id}` en `/conferences/{id}` par exemple), tous les liens doivent être mis à jour manuellement.

- ➔ Utilisez plutôt la fonction Twig `path()` avec le nom de la route

```
# templates/conference/index.html.twig
    <p>
-     <a href="/conference/{{ conference.id }}">View</a>
+     <a href="{{ path('conference', { id: conference.id }) }}>View</a>
    </p>
{% endfor %}
```

La fonction `path()` génère le chemin d'accès vers une page à l'aide du nom de la route. Les valeurs des paramètres dynamiques de la route sont transmises sous la forme d'un objet Twig.

# Paginer les commentaires

Avec des milliers de personnes présentes, on peut s'attendre à un nombre important de commentaires. Si nous les affichons tous sur une seule page, elle deviendra rapidement énorme.

- ▶ Créez une méthode `getCommentPaginator()` dans `CommentRepository`. Cette méthode renvoie un `Paginator` de commentaires basé sur une conférence et un décalage (où commencer)

```
# src/Repository/CommentRepository.php
+use App\Entity\Conference;
+use Doctrine\ORM\Tools\Pagination\Paginator;
...
class CommentRepository extends ServiceEntityRepository
{
+    public const PAGINATOR_PER_PAGE = 2;
...
+    public function getCommentPaginator(Conference $conference, int $offset): Paginator
+    {
+        $query = $this->createQueryBuilder('c')
+            ->andWhere('c.conference = :conference')
+            ->setParameter('conference', $conference)
+            ->orderBy('c.createdAt', 'DESC')
+            ->setMaxResults(self::PAGINATOR_PER_PAGE)
+            ->setFirstResult($offset)
+            ->getQuery()
+        ;
+        ...
+        return new Paginator($query);
+    }
}
```



## Transmettre le Paginator à Twig

Nous avons fixé le nombre maximum de commentaires par page à 2 pour faciliter les tests.

- 💡 Pour gérer la pagination dans le template, transmettez à Twig le Doctrine Paginator au lieu de la Doctrine Collection

```
# src/Controller/ConferenceController.php
+use Symfony\Component\HttpFoundation\Request;
...
- public function show(Environment $twig, Conference $conference, CommentRepository $commentRepository): Response
+ public function show(Request $request, Environment $twig, Conference $conference, CommentRepository $commentRepository): Response
{
    $offset = max(0, $request->query->getInt('offset', 0));
    $paginator = $commentRepository->getCommentPaginator($conference, $offset);
+
    return new Response($twig->render('conference/show.html.twig', [
        'conference' => $conference,
        'comments' => $commentRepository->findBy(['conference' => $conference], ['createdAt' => 'DESC']),
+       'comments' => $paginator,
+       'previous' => $offset - CommentRepository::PAGINATOR_PER_PAGE,
+       'next' => min(count($paginator), $offset + CommentRepository::PAGINATOR_PER_PAGE),
    ]));
}
```

Le contrôleur récupère la valeur du décalage (offset) depuis les paramètres de l'URL (`$request->query`) sous forme d'entier (`getInt()`). Par défaut, sa valeur sera 0 si le paramètre n'est pas défini.

## Gerer la pagination dans Twig

Les décalages précédent et suivant sont calculés sur la base de toutes les informations que nous avons reçues du paginateur.

- ➔ Enfin, mettez à jour le template pour ajouter des liens vers les pages suivantes et précédentes

```
# templates/conference/show.html.twig
{% if comments|length > 0 %}
+   <div>There are {{ comments|length }} comments.</div>
...
+   <p>{{ comment.text }}</p>
{%
+   endfor %}
+   {% if previous >= 0 %}
+     <a href="{{ path('conference', { id: conference.id, offset: previous }) }}">Previous</a>
+   {% endif %}
+   {% if next < comments|length %}
+     <a href="{{ path('conference', { id: conference.id, offset: next }) }}">Next</a>
+   {% endif %}
{%
+   else %}

```

Vous devriez maintenant pouvoir naviguer dans les commentaires avec les liens "Previous" et "Next" :

# Optimiser le contrôleur

Vous avez peut-être remarqué que les deux méthodes présentes dans `ConferenceController` prennent un environnement Twig comme argument.

- ➔ Au lieu de l'injecter dans chaque méthode, appelons la méthode `render()` de la classe parente

```
-use Twig\Environment;

class ConferenceController extends AbstractController
{
    #[Route('/', name: 'homepage')]
-   public function index(Environment $twig, ConferenceRepository $conferenceRepository): Response
+   public function index(ConferenceRepository $conferenceRepository): Response
    {
-       return new Response($twig->render('conference/index.html.twig', [
+       return $this->render('conference/index.html.twig', [
            'conferences' => $conferenceRepository->findAll(),
-           ]);
+           ]);
    }
}
```

# Optimiser le contrôleur

- ▶ Optimisons également la méthode `show()`

```
#Route('/conference/{id}', name: 'conference')
- public function show(Request $request, Environment $twig, Conference $conference, CommentRepository $commentRepository): Response
+ public function show(Request $request, Conference $conference, CommentRepository $commentRepository): Response
{
    $offset = max(0, $request->query->getInt('offset', 0));
    $paginator = $commentRepository->getCommentPaginator($conference, $offset);

    - return new Response($twig->render('conference/show.html.twig', [
+ return $this->render('conference/show.html.twig', [
    'conference' => $conference,
    'comments' => $paginator,
    'previous' => $offset - CommentRepository::PAGINATOR_PER_PAGE,
    'next' => min(count($paginator), $offset + CommentRepository::PAGINATOR_PER_PAGE),
    - ]);
+ ]);
}
```

- ▶ 📄 Commitez notre travail via `git commit -am "Twig"`

## Travaux pratique

Objectif :

- ➔ Modifier la route `/products` pour renvoyer la liste des produits depuis la base de données
- ➔ Créer un template pour la page d'accueil des produits
- ➔ Modifier la route `/product/{id}` pour renvoyer le produit correspondant à l'id depuis la base de données! :
- ➔ Créer un template pour la page d'un produit

# 7. Les événements

# Écouter les événements

Il manque une barre de navigation au layout actuel pour revenir à la page d'accueil ou pour passer d'une conférence à l'autre.

## Ajouter un en-tête au site web

Tout ce qui doit être affiché sur toutes les pages web, comme un en-tête, doit faire partie du layout de base principal :

- ➔ Ajoutez un en-tête au layout de base `templates/base.html.twig`

```
<body>
+   <header>
+     <h1><a href="{{ path('homepage') }}">Guestbook</a></h1>
+     <ul>
+       {% for conference in conferences %}
+         <li><a href="{{ path('conference', { id: conference.id }) }}">{{ conference }}</a></li>
+       {% endfor %}
+     </ul>
+     <hr />
+   </header>
+   {% block body %}{% endblock %}
</body>
```

L'ajout de ce code au layout signifie que tous les templates qui l'étendent doivent définir une variable `conferences`, créée et transmise par leurs contrôleurs.

# Transmettre des variables à tous les templates

Comme nous n'avons que deux contrôleurs, vous pourriez procéder comme ceci (**ne modifiez pas votre code car nous verrons très vite une meilleure façon de faire**) :

```
# #[Route('/conference/{id}', name: 'conference')]
- public function show(Request $request, Conference $conference, CommentRepository $commentRepository): Response
+ public function show(Request $request, Conference $conference, CommentRepository $commentRepository, ConferenceRepository $conferenceRepository): Response
{
    $offset = max(0, $request->query->getInt('offset', 0));
    $paginator = $commentRepository->getCommentPaginator($conference, $offset);

    return $this->render('conference/show.html.twig', [
        'conferences' => $conferenceRepository->findAll(),
        ...
    ]);
}
```

🤔 Imaginez devoir mettre à jour des dizaines de contrôleurs. Et faire la même chose sur tous les nouveaux. Ce n'est pas très pratique. Il doit y avoir un meilleur moyen.

**Twig a la notion de variables globales.** Une variable globale est disponible dans tous les templates générés. Vous pouvez les définir dans un fichier de configuration, mais cela ne fonctionne que pour les valeurs statiques. Pour ajouter toutes les conférences comme variable globale Twig, nous allons créer un "**listener**".



# Découvrir les événements Symfony

Symfony intègre un composant Event Dispatcher.

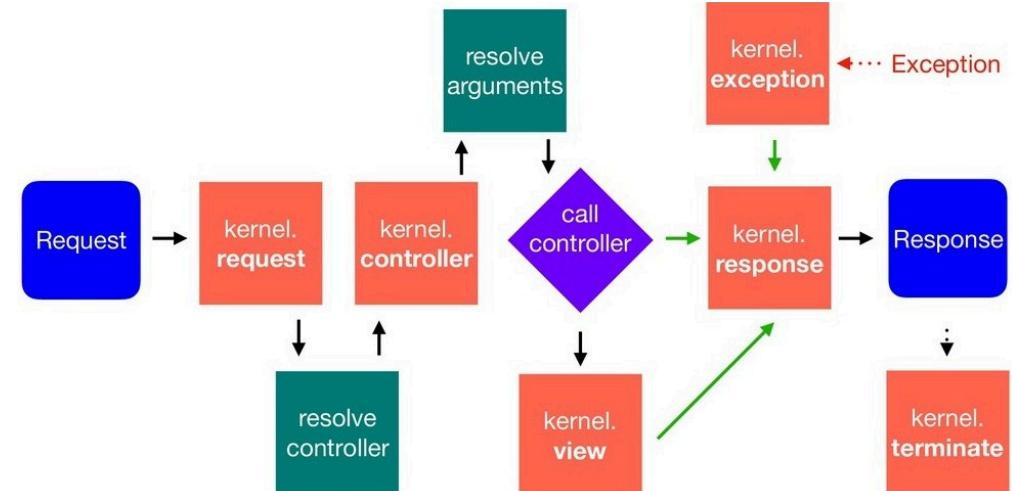
- 👉 Un dispatcher répartit certains événements à des moments précis que les listeners peuvent écouter.
- 👉 Les listeners sont des hooks dans le cœur du framework.

Par exemple:

Certains événements vous permettent d'interagir avec le **cycle de vie des requêtes HTTP**. Pendant le traitement d'une requête, le dispatcher répartit les événements :

- Lorsqu'une requête a été créée,
- Lorsqu'un contrôleur est sur le point d'être exécuté
- Lorsqu'une réponse est prête à être envoyée
- Ou lorsqu'une exception a été levée.

👉 Un listener peut écouter un ou plusieurs événements et exécuter une logique basée sur le contexte de l'événement.



## Les événements dans Symfony

Les événements sont des points d'extension bien définis qui rendent le framework plus générique et extensible. De nombreux composants Symfony tels que Security, Messenger, Workflow ou Mailer les utilisent largement.

**Un autre exemple intégré d'événements et de listeners en action est le cycle de vie d'une commande :**

- 👉 Vous pouvez créer un listener pour exécuter du code avant n'importe quelle commande.

Tout paquet ou bundle peut également déclencher ses propres événements pour rendre son code extensible.



# Implémenter un subscriber

Pour éviter d'avoir un fichier de configuration (`service.yaml`) qui décrit les événements qu'un listener veut écouter, utilisons plutôt un subscriber.

Un **subscriber** est un **listener** avec une méthode statique `getSubscribedEvents()` qui retourne sa configuration. Ceci permet aux subscribers d'être enregistrés automatiquement dans le **dispatcher Symfony**.

Résumons donc la différence entre un **listener** et un **subscriber** :

- 👉 Un **listener** est une classe avec une méthode publique qui écoute un événement spécifique.
- 👉 Un **subscriber** est une classe avec une méthode statique qui retourne la configuration de tous les événements qu'il écoute.

Vous connaissez la chanson par cœur maintenant

- ▶ Utilisez le **Maker Bundle** pour générer un subscriber :

```
symfony console make:subscriber TwigEventSubscriber
```

La commande vous demande quel événement vous voulez écouter.

- ▶ Choisissez l'événement `Symfony\Component\HttpKernel\Event\ControllerEvent` qui est envoyé juste avant l'appel d'un contrôleur.

C'est le meilleur moment pour injecter la variable globale `conferences` afin que Twig y ait accès lorsque le contrôleur générera le template.

# Personnaliser le subscriber généré

- ▶ Mettez votre subscriber à jour comme suit :

```
class TwigEventSubscriber implements EventSubscriberInterface
{
    public function __construct(
        private Environment $twig,
        private ConferenceRepository $conferenceRepository
    ) { }

    public function onControllerEvent(ControllerEvent $event): void
    {
        // ...
        $this->twig->addGlobal('conferences', $this->conferenceRepository->findAll());
    }
}
```

Maintenant, vous pouvez ajouter autant de contrôleurs que vous le souhaitez : la variable `conferences` sera toujours disponible dans Twig.



Nous parlerons d'une alternative bien plus performante dans une prochaine étape.

# Trier les conférences par année et par ville

Le tri de la liste des conférences par année peut faciliter la navigation. Nous pourrions créer notre propre méthode pour récupérer et trier toutes les conférences.

- ➡ Mais nous allons plutôt remplacer l'implémentation par défaut de la méthode `CommentRepository::findAll()`, afin que le tri s'applique partout :

```
+ public function findAll(): array
+ {
+     return $this->findBy([], ['year' => 'ASC', 'city' => 'ASC']);
+ }
+
public function save(Conference $entity, bool $flush = false): void
```

À la fin de cette étape, le site web devrait ressembler à ceci :

Guestbook

: Amsterdam 2019  
: Paris 2020

---

Give your feedback!

Amsterdam 2019  
[View](#)

Paris 2020  
[View](#)

- ➡ Commitez notre travail via `git commit -am "Écouter les événements"`

# Définir des lifecycle callbacks

Lors de la création d'un nouveau commentaire, ce serait bien si la date `createdAt` était automatiquement définie à la date et à l'heure courantes.

Doctrine a différentes façons de manipuler les objets et leurs propriétés pendant leur cycle de vie (avant la création de la ligne dans la base de données, après la mise à jour de la ligne, etc.).

Lorsque le comportement n'a besoin d'aucun service et ne doit être appliqué qu'à un seul type d'entité.

- ➔ Définissez un callback dans la classe entité :

```
#[ORM\Entity(repositoryClass: CommentRepository::class)]
+#[ORM\HasLifecycleCallbacks]
class Comment
{
    @@ -91,8 +92,12 @@ class Comment

+    #[ORM\PrePersist]
+    public function setCreatedAtValue() : static
+    {
+        $this->createdAt = new \DateTimeImmutable();
+        return $this;
+    }
+
    public function getConference(): ?Conference
```

L'événement `ORM\PrePersist` est déclenché lorsque l'objet est enregistré dans la base de données pour la toute première fois. Lorsque cela se produit, la méthode `setCreatedAtValue()` est appelée et la date et l'heure courantes sont utilisées pour la valeur de la propriété `createdAt`.

- ➔ Ajoutez un commentaire depuis l'interface d'administration et vérifiez que la date est correcte.

## Ajouter des slugs aux conférences

Les URLs des conférences n'ont pas de sens : `/conference/1`. Plus important encore, ils dépendent d'un détail d'implémentation

" la clé primaire de la base de données est révélée "

Pourquoi ne pas plutôt utiliser des URLs telles que `/conference/paris-2020`? Ce serait plus joli. `paris-2020`, c'est ce que l'on appelle le slug de la conférence.

- ➔ Ajoutez une nouvelle propriété `slug` pour les conférences (une chaîne non nulle de 255 caractères) :

```
symfony console make:entity Conference
```

La commande vous demande le nom de la propriété. Entrez `slug` et choisissez `string` comme type.

# Applicons la migration

- ➔ Créez un fichier de migration pour ajouter la nouvelle colonne et Exécutez cette nouvelle migration.
- ! Vous avez une erreur ? C'était prévu. Pourquoi ?
- 👉 Parce que nous avons demandé que le `slug` ne soit pas `null`, et que les entrées existantes dans la base de données de la conférence obtiendront une valeur `null` lorsque la migration sera exécutée. Corrigeons cela en ajustant la migration.

```
public function up(Schema $schema): void
{
    // this up() migration is auto-generated, please modify it to your needs
    - $this->addSql('ALTER TABLE conference ADD slug VARCHAR(255) NOT NULL');
    + $this->addSql('ALTER TABLE conference ADD slug VARCHAR(255)');
    + $this->addSql("UPDATE conference SET slug=CONCAT(LOWER(city), '-', year)");
    + $this->addSql('ALTER TABLE conference ALTER COLUMN slug SET NOT NULL');
}
```

L'astuce ici est d'ajouter la colonne et de lui permettre d'être `null`, puis de définir une valeur non `null` pour le `slug`, et enfin, de changer la colonne de `slug` pour ne plus permettre `null`.

La migration devrait fonctionner maintenant.

- ➔ Exécutez la migration

# Assurer l'unicité des slugs

Étant donné que l'application utilisera bientôt les slugs pour trouver chaque conférence, ajustons l'entité `Conference` pour s'assurer que les valeurs des slugs soient uniques dans la base de données :

- ➔ Ajoutez une contrainte d'unicité sur la propriété `slug` de la classe `Conference` :

```
+use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;  
  
#[ORM\Entity(repositoryClass: ConferenceRepository::class)]  
+#[UniqueEntity('slug')]  
class Conference  
{  
    #[ORM\Id]  
  
    @@ ...  
  
    #[ORM\OneToMany(mappedBy: 'conference', targetEntity: Comment::class, orphanRemoval: true)]  
    private Collection $comments;  
  
-    #[ORM\Column(length: 255)]  
+    #[ORM\Column(length: 255, unique: true)]  
    private ?string $slug = null;
```

👑 Comme vous l'aurez deviné, nous devons exécuter la danse de la migration

- ➔ Créez une nouvelle migration et exécutez-la.

# Générer des slugs

Générer un **slug** qui se lit bien dans une URL (où tout ce qui n'est pas des caractères **ASCII** doit être encodé) est une tâche difficile, surtout pour les langues autres que l'anglais. Comment convertir é en e par exemple ?

Au lieu de réinventer la roue, utilisons le composant **Symfony String**, qui facilite la manipulation des chaînes et fournit un **slugger**.

- ▶ Dans la classe Conference, ajoutez une méthode `computeSlug()`, qui calcule le slug en fonction des données de la conférence :

```
+use Symfony\Component\String\Slugger\SluggerInterface;  
  
#[ORM\Entity(repositoryClass: ConferenceRepository::class)]  
  
@@ ...  
  
+ public function computeSlug(SluggerInterface $slugger) : static  
+ {  
+     if (!$this->slug || '-' === $this->slug) {  
+         $this->slug = (string) $slugger->slug((string) $this)->lower();  
+     }  
+     return $this;  
+ }  
+  
public function getCity(): ?string
```

La méthode `computeSlug()` ne calcule un slug que lorsque le slug courant est vide ou défini à la valeur spéciale.

💡 Pourquoi avons-nous besoin de cette valeur particulière ?

👉 Parce que lors de l'ajout d'une conférence dans l'interface d'administration, le slug est nécessaire. Nous avons donc besoin d'une valeur non vide qui indique à l'application que nous voulons que le slug soit généré automatiquement.

## Définir un lifecycle callback complexe

Comme pour la propriété `createdAt`, la propriété `slug` doit être définie automatiquement à chaque fois que la conférence est mise à jour en appelant la méthode `computeSlug()`.

Mais comme cette méthode dépend d'une implémentation de `SluggerInterface`, nous ne pouvons pas ajouter un événement `prePersist` comme avant (nous n'avons pas la possibilité d'injecter le slugger).

- ➡ Créez plutôt une classe de listener d'entité Doctrine [src/EntityListener/ConferenceEntityListener.php](#)

```
namespace App\EntityListener;

use App\Entity\Conference;
use Doctrine\ORM\Event\LifecycleEventArgs;
use Symfony\Component\String\Slugger\SluggerInterface;

class ConferenceEntityListener
{
    public function __construct(private SluggerInterface $slugger) { }

    public function __invoke(Conference $conference, LifecycleEventArgs $event) : void
    {
        $conference->computeSlug($this->slugger);
    }
}
```

Notez que le slug est modifié lorsqu'une nouvelle conférence est créée et lorsqu'elle est mise à jour.

# Le conteneur d'injection de dépendance

⚠️ Jusqu'à présent, nous n'avons pas parlé d'un élément clé de Symfony, "*le conteneur d'injection de dépendance*".

- 👉 Le conteneur est responsable de la gestion des services : leur création, et leur injection en cas de besoin.
- 👉 Un service est un objet "global" qui fournit des fonctionnalités (par exemple un **mailer**, un **logger**, un **slugger**, etc.) contrairement aux objets de données (par exemple les instances d'entités Doctrine `App\Entity`).
- 👉 Vous interagissez rarement directement avec le conteneur, car il injecte automatiquement des objets de service quand vous en avez besoin.

Par exemple, le conteneur injecte les objets en arguments du contrôleur lorsque vous les typez, comme nous l'avons fait avec `Environment` et `ConferenceRepository` dans la méthode `index()` du contrôleur `ConferenceController`.

```
class ConferenceController extends AbstractController
{
    #[Route('/', name: 'homepage')]
    public function index(ConferenceRepository $conferenceRepository): Response
    {
        // ...
    }
}
```

# Configurer un service dans le conteneur

🧐 Si vous vous demandez comment le listener d'événement a été initialisé à l'étape précédente, vous avez maintenant la réponse :

"Le conteneur"

Lorsqu'une classe implémente des **interfaces spécifiques**, ⇒ le conteneur sait que la classe doit être initialisée d'une certaine manière.

Dans ce cas précis, puisque notre classe `ConferenceEntityListener` n'implémente aucune interface et n'étend d'aucune autre classe, Symfony ne peut pas la configurer automatiquement.

- ➡ Utilisons un attribut pour l'aider :

```
+use Doctrine\Bundle\DoctrineBundle\Attribute\AsEntityListener;
+use Doctrine\ORM\Events;

#[AsEntityListener(event: Events::prePersist, entity: Conference::class)]
#[AsEntityListener(event: Events::preUpdate, entity: Conference::class)]
class ConferenceEntityListener
{
```

L'attribut `AsEntityListener` indique au conteneur que la classe `ConferenceEntityListener` doit être initialisée comme un listener d'événement pour les événements `prePersist` et `preUpdate` de l'entité `Conference`.

! Ne confondez pas les **listeners d'événements Doctrine** et **ceux de Symfony**. Même s'ils se ressemblent beaucoup, ils n'utilisent pas la même infrastructure en interne.

# Utiliser des slugs dans l'application

-  **Essayez d'ajouter d'autres conférences dans l'interface d'administration et changez la ville ou l'année d'une conférence existante.**  
Le slug ne sera pas mis à jour sauf si vous utilisez la valeur spéciale "-"
-  **La dernière modification consiste à mettre à jour les contrôleurs et les modèles pour utiliser le slug de la conférence pour les routes, au lieu de son id :**

```
# src/Controller/ConferenceController.php
-  #[Route('/conference/{id}', name: 'conference')]
+  #[Route('/conference/{slug}', name: 'conference')]
  public function show(Request $request, Conference $conference, CommentRepository $commentRepository): Response

# templates/base.html.twig
    {% for conference in conferences %}
-      <li><a href="{{ path('conference', { id: conference.id }) }}>{{ conference }}</a></li>
+      <li><a href="{{ path('conference', { slug: conference.slug }) }}>{{ conference }}</a></li>
    {% endfor %}

# templates/conference/index.html.twig
    <p>
-      <a href="{{ path('conference', { id: conference.id }) }}>View</a>
+      <a href="{{ path('conference', { slug: conference.slug }) }}>View</a>
    </p>

# templates/conference/show.html.twig
    {% if previous >= 0 %}
-      <a href="{{ path('conference', { id: conference.id, offset: previous }) }}>Previous</a>
+      <a href="{{ path('conference', { slug: conference.slug, offset: previous }) }}>Previous</a>
    {% endif %}
    {% if next < comments|length %}
-      <a href="{{ path('conference', { id: conference.id, offset: next }) }}>Next</a>
+      <a href="{{ path('conference', { slug: conference.slug, offset: next }) }}>Next</a>
    {% endif %}
```

# Naviguer avec les slugs

L'accès à la page d'une conférence devrait maintenant se faire grâce à son slug : [/conference/paris-2020](#).

## Guestbook

- [Amsterdam 2019](#)
- [Paris 2020](#)

### Amsterdam 2019 Conference

There are 4 comments.



Lucas

Dec 7, 2022, 9:10 AM

That was an amazing conference.



- ➔ 📜 Commitez notre travail via `git commit -am "Gérer le cycle de vie des objets Doctrine"`

## Travaux pratique

1. Faire le Quizz : Symfony Part 7

2. Objectif :

- ➔ Ajouter une propriété `createdAt` et `updatedAt` à toutes les entités
- ➔ Ajouter un listener pour mettre à jour les propriétés `updatedAt` et `createdAt`
  - A savoir que la propriété `updatedAt` doit être mise à jour à chaque modification de l'entité
  - La propriété `createdAt` doit être mise à jour à la création de l'entité
- ➔ Bonus utilisez le trait pour factoriser le code

# 8. Les formulaires

## Générer un form type

Il est temps de permettre aux personnes présentes de donner leur avis sur les conférences. Elles feront part de leurs commentaires au moyen d'un formulaire HTML.

- ➔ Utilisez le Maker Bundle pour générer une classe de formulaire :

```
symfony console make:form CommentFormType Comment
```

La classe `App\Form\CommentFormType` a été générée et définit un formulaire pour l'entité `App\Entity\Comment`

Un **form type** décrit les champs de formulaire liés à un modèle. Il effectue la conversion des données entre les données soumises et les propriétés de la classe de modèle.

Par défaut, Symfony utilise les métadonnées de l'entité `Comment`, comme les métadonnées Doctrine, pour deviner la configuration de chaque champ, avec `data_class` défini sur `Comment::class` dans la méthode `configureOptions()`. Mais nous pourrions utiliser un modèle différent si nous le souhaitons.

Par exemple, le champ `text` se présente sous la forme d'un `textarea` parce qu'il utilise une colonne plus grande dans la base de données. En effet, on lui a défini le type de column doctrine `Types::TEXT` dans l'entité `Comment`.

# Transmettre le formulaire au template

- ▶ Pour afficher le formulaire, créez-le dans le contrôleur et transmettez-le au template :

```
+use App\Entity\Comment;
use App\Entity\Conference;
+use App\Form\CommentFormType;

@@ ...

#[Route('/conference/{slug}', name: 'conference')]
public function show(Request $request, Conference $conference, CommentRepository $commentRepository): Response
{
    $comment = new Comment();
    $form = $this->createForm(CommentFormType::class, $comment);
    ...

@@ ...

    'next' => min(count($paginator), $offset + CommentRepository::PAGINATOR_PER_PAGE),
    'comment_form' => $form,
```

Vous ne devriez jamais instancier directement le form type.

- 👉 Utilisez plutôt la méthode `createForm()`. Cette méthode fait partie d'`AbstractController` et facilite la création de formulaires.

💡 Pour transmettre le formulaire au template, nous l'ajoutons à la liste des variables transmises au template. Il n'est plus nécessaire d'utiliser la méthode `createView()` pour obtenir l'objet form adapté au template.

# Afficher un formulaire

- ➡ L'affichage du formulaire dans le template peut se faire via la fonction Twig `form` dans le template `templates/conference/show.html.twig`:

```
% endif %}
+
<h2>Add your own feedback</h2>
+
{{ form(comment_form) }}
{% endblock %}
```

Lorsque vous rafraîchissez la page d'une conférence dans le navigateur, notez que chaque champ de formulaire affiche la balise HTML appropriée (le type de données est défini à partir du modèle) :

La fonction `form()` génère le formulaire HTML en fonction de toutes les informations définies dans le `form type`. Elle ajoute également `enctype=multipart/form-data` à la balise `<form>` comme l'exige le champ d'upload de fichier. De plus, elle se charge d'afficher les messages d'erreur lorsque la soumission comporte des erreurs. Tout peut être personnalisé en remplaçant les templates par défaut, mais nous n'en aurons pas besoin pour ce projet.

Dec 7, 2022, 9:11 AM

That was a great conference.



Thomas

Dec 7, 2022, 9:11 AM

That was a cool conference.

[Next](#)

## Add your own feedback

Author

Text

Email

Created at

Jan 1 2017  
00:00

Photo filename

Conference Amsterdam 2019

# Personnaliser un form type

Même si les champs de formulaire sont configurés en fonction de leur modèle associé, vous pouvez personnaliser la configuration par défaut directement dans la classe de form type :

- ➔ Modifiez la classe `CommentFormType` pour personnaliser certains champs :

```
class CommentFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('author')
            +->add('author', null, [
            +    'label' => 'Your name',
            +])
            ->add('text')
            ->add('email')
            ->add('createdAt')
            ->add('photoFilename')
            ->add('conference')
            +->add('email', EmailType::class)
            +->add('photo', FileType::class, [
            +    'required' => false,
            +    'mapped' => false,
            +    'constraints' => [
            +        new Image(['maxSize' => '1024k'])
            +    ],
            +])
            ->add('submit', SubmitType::class)
    ;
}
```

Notez que nous avons ajouté un bouton `submit` (qui nous permet de continuer à utiliser simplement  `{{ form(comment_form) }}`  dans le template).

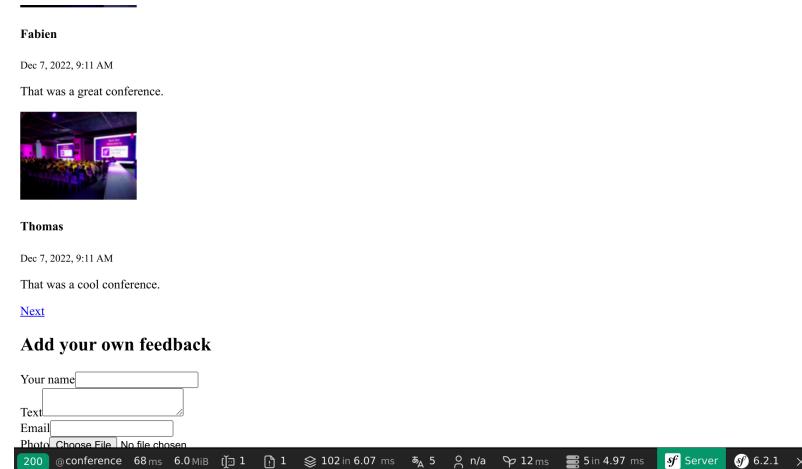
# Gerer les champs non mappés

Certains champs ne peuvent pas être auto-configurés, comme par exemple `photoFilename`.

- 👉 L'entité `Comment` n'a besoin d'enregistrer que le nom du fichier photo, **mais le formulaire doit s'occuper de l'upload du fichier lui-même.**

Pour traiter ce cas, nous avons ajouté un champ appelé `photo` qui est un champ "**non mapped**" : il ne sera associé à aucune propriété de `Comment`. Nous le gérerons manuellement pour implémenter une logique spécifique (comme l'upload de la photo sur le disque).

- ➡ Rafraîchissez la page d'une conférence dans le navigateur, et vous verrez que le formulaire a été mis à jour :



Comme autre exemple de personnalisation, nous avons également modifié le libellé par défaut de certains champs.

# Valider des modèles

- 💡 Pour vérifier que les données soumises sont valides, le formulaire utilise le type de champ `email` pour l'email du commentaire et définit la plupart des champs en `required`. Notez qu'il contient également un champ `_token` caché pour nous protéger des attaques `CSRF`.
- 💻 Mais si la soumission du formulaire contourne la validation HTML (en utilisant un client HTTP comme cURL, qui n'applique pas ces règles de validation), des données invalides peuvent atteindre le serveur. 😱

- ➡ Pour éviter cela, nous devons ajouter des contraintes de validation à l'entité `Comment` :

```
use Doctrine\ORM\Mapping as ORM;
+use Symfony\Component\Validator\Constraints as Assert;

@@ ...

+  #[Assert\NotBlank]
+  #[Assert\Length(max: 255)]
private ?string $author = null;

#[ORM\Column(type: Types::TEXT)]
+  #[Assert\NotBlank]
private ?string $text = null;

#[ORM\Column(length: 255)]
+  #[Assert\NotBlank]
+  #[Assert\Length(max: 255)]
+  #[Assert\Email]
private ?string $email = null;
```

Nous utilisons ici les **contraintes de validation de Symfony** pour définir les règles de validation. Par exemple, le champ `author` ne doit pas être vide, le champ `email` doit être une adresse email valide, etc.

# Gérer la soumission d'un formulaire

Le code que nous avons écrit jusqu'à présent est suffisant pour afficher le formulaire et ses messages d'erreur.

Nous devrions maintenant nous occuper de la soumission du formulaire et de la persistance de ses informations dans la base de données depuis le contrôleur :

- **Modifiez la méthode `show()` du contrôleur pour gérer la soumission du formulaire :**

```
class ConferenceController extends AbstractController
{
+    public function __construct(private readonly EntityManagerInterface $entityManager) { }
@@ ...
        $form = $this->createForm(CommentFormType::class, $comment);
+        $form->handleRequest($request);
+        if ($form->isSubmitted() && $form->isValid()) {
+            $comment->setConference($conference);
+
+            $this->entityManager->persist($comment);
+            $this->entityManager->flush();
+
+            return $this->redirectToRoute('conference', ['slug' => $conference->getSlug()]);
+        }
    }
}
```

## Essayer le formulaire

Faisons un récapitulatif de ce que nous avons fait jusqu'à présent :

- Lorsque le formulaire est soumis, l'objet `Comment` est mis à jour en fonction des données soumises. Puis redirection vers la conférence qui doit être la même que celle de l'URL (nous l'avons supprimée du formulaire).
- Si le formulaire n'est pas valide, nous affichons la page, mais le formulaire contiendra maintenant les valeurs soumises et les messages d'erreur afin qu'ils puissent être affichés à l'internaute.
-  **Essayez le formulaire.**
  -  Il devrait fonctionner correctement et les données devraient être stockées dans la base de données (vérifiez-les dans l'interface d'administration).
  -  Il y a cependant un problème : les photos. Elles ne fonctionnent pas puisque nous ne les avons pas encore traitées dans le contrôleur. Choses que nous allons faire dans la prochaine étape.

## Définition du chemin de stockage des photos

Les photos uploadées doivent **être stockées sur le disque local**, à un endroit accessible par un navigateur afin que nous puissions les afficher sur la page d'une conférence.

- 👉 Nous les stockerons dans le dossier `public/uploads/photos` :

Comme nous ne souhaitons pas mettre le répertoire en dur dans le code, nous devons trouver un moyen de le stocker de façon globale. **Le conteneur Symfony** est capable de stocker des paramètres (`parameters`) en plus des services pour permettre de les configurer :

- ➡ Ajoutez un paramètre `photo_dir` dans le fichier `config/services.yaml` :

```
# config/services.yaml
parameters:
+   photo_dir: "%kernel.project_dir%/public/uploads/photos"
```

## Utiliser le chemin dans le contrôleur

Nous avons déjà vu comment les services sont automatiquement injectés dans les arguments des constructeurs. Pour les paramètres du conteneur, nous pouvons les injecter explicitement en utilisant l'attribut `Autowire`.

Maintenant, nous avons tout ce qu'il nous faut pour implémenter la logique nécessaire au stockage du fichier soumis sous sa destination finale.

- ➔ Modifiez la signature de méthode `show()` du contrôleur pour récupérer le paramètre `photo_dir` :

```
# src/Controller/ConferenceController.php
- public function show(Request $request, Conference $conference, CommentRepository $commentRepository): Response
-
{
+ public function show(
+     Request $request,
+     Conference $conference,
+     CommentRepository $commentRepository,
+     #[Autowire('%photo_dir%')]
+     string $photoDir,
+ ): Response {
```

# Gérer l'uploader des fichiers

-  **Modifiez la méthode `show()` du contrôleur pour gérer l'upload de la photo :**

```
# src/Controller/ConferenceController.php

if ($form->isSubmitted() && $form->isValid()) {
    $comment->setConference($conference);

    if ($photo = $form['photo']->getData()) {
        $filename = bin2hex(random_bytes(6)) . '.' . $photo->guessExtension();
        try {
            $photo->move($photoDir, $filename);
        } catch (FileException $e) {
            // unable to upload the photo, give up
        }
        $comment->setPhotoFilename($filename);
    }

    $this->entityManager->persist($comment);
    $this->entityManager->flush();
}
```

- Pour gérer les uploads de photos, nous créons un nom aléatoire pour le fichier.
- Ensuite, nous déplaçons le fichier uploadé à son emplacement final (le répertoire photo).
- Enfin, nous stockons le nom du fichier dans l'objet Comment.

Notre code est maintenant capable de gérer les uploads de photos.

-  **Essayez d'uploader un fichier PDF au lieu d'une photo.**

 Vous devriez voir les messages d'erreur en action. Déboguons-les dans la prochaine étape.

# Déboguer des formulaires

Lorsqu'un formulaire est soumis et que quelque chose ne fonctionne pas correctement, utilisez le panneau "Form" du **Symfony Profiler**. Il vous donne des informations sur le formulaire, toutes ses options, les données soumises et comment elles sont converties en interne. Si le formulaire contient des erreurs, elles seront également répertoriées.

Le workflow classique d'un formulaire est le suivant :

- Le formulaire est affiché sur une page ;
- L'internaute soumet le formulaire via une requête **POST** ;
- Le serveur redirige l'internaute, soit vers une autre page, soit vers la même page.

Mais comment pouvez-vous accéder au profileur pour une requête de soumission réussie ? Étant donné que la page est immédiatement redirigée, nous ne voyons jamais la barre d'outils de débogage Web pour la requête **POST**.

Pas de problème : sur la page redirigée, survolez la partie verte "200" à gauche. Vous devriez voir la redirection "302" avec un lien vers le profileur (entre parenthèses).

The screenshot shows the Symfony Profiler interface with the 'Forms' panel selected. At the top, it displays a POST request to 'https://127.0.0.1:8000/conference/amsterdam-2019'. The status is '302 Found'. On the left, there are filters for 'IP', 'Method' (set to 'Any'), 'Status', 'URL', 'Token', 'From' (date range), 'Until' (date range), and 'Results' (set to 10). Below these filters is a 'Request / Response' section with tabs for 'Performance', 'Validator', and 'Forms' (which is currently active). The 'Forms' tab shows a single entry: 'comment\_form' (Form type: "App\Form\CommentFormType"). To the right of the form entry, under the 'comment\_form' heading, it says 'This form has no errors.' The sidebar on the left also lists other sections like 'Exception', 'Logs', 'Events', 'Routing', 'Cache', and 'Translation'.

# Gérer les photos depuis l'interface d'administration

L'interface d'administration affiche actuellement le nom du fichier photo, mais nous voulons voir la vraie photo :

- ➔ Modifiez la classe `CommentCrudController` pour afficher la photo :

```
use EasyCorp\Bundle\EasyAdminBundle\Field\TextareaField;
+use EasyCorp\Bundle\EasyAdminBundle\Field\ImageField;

@@ ...

-    yield TextField::new('photoFilename')
+    yield ImageField::new('photoFilename')
+        ->setUploadDir('/public/uploads/photos')
+        ->setUploadedFileNamePattern(fn(UploadedFile $photo) => Comment::setFilename($photo))
+        ->setBasePath('/uploads/photos')
+        ->setLabel('Photo')
->onlyOnIndex();
```

- ➔ Ajoutez une méthode statique `setFilename()` à la classe `Comment` afin de factoriser la logique de génération de nom de fichier :

```
public static function setFilename(UploadedFile $photo): string
{
    return bin2hex(random_bytes(6)) . '.' . $photo->guessExtension();
}
```

## Exclure les photos uploadées de Git

Ne commitez pas encore ! Nous ne voulons pas stocker les images uploadées dans le dépôt Git. Ajoutez le dossier `/public/uploads` au fichier `.gitignore` :

- ➔ Ajoutez `/public/uploads` au fichier `.gitignore` :

```
+/public/uploads  
###> symfony/framework-bundle ###
```

- ➔ 📄 Commitez notre travail via

```
git add . && git commit -m "Formulaire"
```

## 9. Sécurité



## Sécuriser l'interface d'admin

👉 L'interface d'administration ne doit être accessible que par des personnes autorisées. La sécurisation de cette zone du site peut se faire à l'aide du composant **Symfony Security**.

Même si les internautes ne pourront pas créer leur propre compte sur le site, nous allons créer un système d'authentification entièrement fonctionnel pour l'admin.

👉 La première étape consiste à définir une entité `User`. Pour éviter toute confusion, nommons-la plutôt `Admin`.

Pour utiliser l'entité `Admin` dans le système d'authentification de Symfony, celle-ci doit respecter certaines exigences spécifiques. Par exemple, elle a besoin d'une propriété `password`.



# Définir une entité User

- ➔ Utilisez la commande dédiée `make:user` pour créer l'entité `Admin` au lieu de la commande traditionnelle `make:entity`

```
symfony console make:user Admin
```

- ➔ Répondez aux questions qui vous sont posées :

- Nous voulons utiliser Doctrine pour stocker nos users (`yes`)
- Utiliser `username` pour le nom d'affichage unique des admins
- Et chaque admin aura un mot de passe (`yes`).

La commande a généré une classe `App\Entity\Admin` avec les propriétés `username` et `password`. Elle contient des aussi des méthodes comme `getRoles()`, `eraseCredentials()` et d'autres qui sont nécessaires au système d'authentification de Symfony.

 Si vous voulez ajouter d'autres propriétés à l'entité Admin, exécutez `make:entity`.

- ➔ Implementer l'interface `\Stringable` dans l'entité `Admin` et ajouter notre nouvelle entité à Easyadmin :

```
- class Admin implements UserInterface, PasswordAuthenticatedUserInterface
+ class Admin implements UserInterface, PasswordAuthenticatedUserInterface, \Stringable
{
@@...
+    public function __toString(): string
+    {
+        return $this->username;
+    }
}
```

# Configuration de la sécurité

En plus de générer l'entité `Admin`, la commande `make:user` a également mis à jour la configuration de sécurité pour connecter l'entité au système d'authentification :

```
# config/packages/security.yaml

providers:
- users_in_memory: { memory: null }
+ app_user_provider:
+   entity:
+     class: App\Entity\Admin
+     property: username
firewalls:
  main:
    lazy: true
- provider: users_in_memory
+ provider: app_user_provider
```

💡 Nous laissons Symfony choisir le meilleur algorithme possible pour hacher les mots de passe (il évoluera avec le temps).

```
password_hashers:
  Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

- ➡ Il est temps de générer une migration et de migrer la base de données :

## Générer un mot de passe pour l'admin

Nous ne développerons pas de système dédié pour créer des comptes d'administration. Encore une fois, nous **n'aurons qu'un seul admin**. Le login sera admin et nous devons générer le hash du mot de passe.

- ▶ Lancer la commande `security:hash-password` pour générer le hash du mot de passe

```
symfony console security:hash-password myPassword 'App\User\Admin'
```

! Notez le hash généré, nous en aurons besoin pour insérer l'admin dans la base de données.

- ▶ Ajouter le CRUD Admin de la classe `Admin` dans easyadmin et définissez-y le contenu de la méthode `crudFields` comme ce qui suit:

```
public function configureFields(string $pageName): iterable
{
    yield IdField::new('id')->hideOnForm();
    yield TextField::new('username');
    yield TextField::new('password');
    yield ArrayField::new('roles');
}
```

# Configurer le système d'authentification

Maintenant que nous avons un admin, nous pouvons sécuriser l'interface d'administration. Symfony accepte plusieurs stratégies d'authentification. Utilisons un classique système d'authentification par formulaire.

- ➔ Exécutez la commande `make:security`

- pour mettre à jour la configuration de sécurité
- générer un template pour la connexion
- créer une classe d'authentification (`Authenticator`) :

```
symfony console make:security
```

La commande vous pose quelques questions :

- Sélectionnez **1** pour générer une classe d'authentification pour le formulaire de connexion
- Nommez la classe d'authentification `AppAuthenticator`
- Le contrôleur `SecurityController`
- Créez une URL `/logout` (yes).

La commande a mis à jour la configuration de sécurité `config/packages/security.yaml` pour lier les classes générées :

```
main:  
    lazy: true  
    provider: app_user_provider  
+    custom_authenticator: App\Security\AppAuthenticator  
+    logout:  
+        path: app_logout  
+        # where to redirect after logout  
+        # target: app_any_route
```

# Ajouter les règles de contrôle d'accès

Comme l'indique la sortie de la commande `make:auth`.

- ▶ Nous devons personnaliser la route dans la méthode `onAuthenticationSuccess()` pour rediriger l'admin lorsqu'il a réussi à se connecter :

```
# src/Security/AppAuthenticator.php
-
-    // For example:
-    // return new RedirectResponse($this->urlGenerator->generate('some_route'));
-    throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
+
+    return new RedirectResponse($this->urlGenerator->generate('admin'));
```

Un système de sécurité se compose de deux parties : l'authentification et l'autorisation. Lors de la création de l'admin, nous lui avons donné le rôle `ROLE_ADMIN`.

- ▶ Limitons la section `/admin` aux seules personnes ayant ce rôle en ajoutant une règle à `access_control` :

```
# config/packages/security.yaml
access_control:
-
-    # - { path: ^/admin, roles: ROLE_ADMIN }
+
+    - { path: ^/admin, roles: ROLE_ADMIN }
```

Les règles `access_control` limitent l'accès par des expressions régulières. Lorsqu'une personne connectée tente d'accéder à une URL qui commence par `/admin`, le système de sécurité vérifie qu'elle a bien le rôle `ROLE_ADMIN`.

# S'authentifier avec le formulaire de connexion

Si vous essayez d'accéder à l'interface d'administration, vous devriez maintenant être redirigé vers la page de connexion et être invité à entrer un identifiant et un mot de passe :

- ➔ Connectez-vous en utilisant `admin` et le mot de passe que vous avez choisi précédemment.

Si vous avez copié exactement ma requête SQL, le mot de passe est `admin`.

 Notez qu'EasyAdmin s'intègre automatiquement au système d'authentification de Symfony :

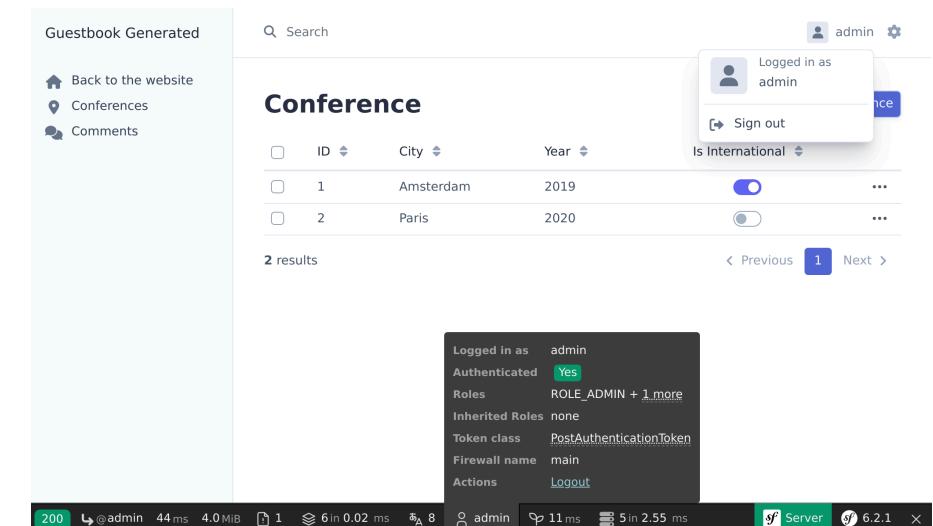
 Si vous voulez créer un système complet d'authentification par formulaire, jetez un coup d'œil à la commande `make:registration-form`.

## Guestbook

- [Amsterdam 2019](#)
- [Paris 2020](#)

## Please sign in

Username  Password



The screenshot shows the EasyAdmin interface for managing conferences. At the top right, there is a user menu with the option "Logged in as admin". Below it, the "Conference" list page is displayed, showing two entries: one for Amsterdam in 2019 and one for Paris in 2020. The interface includes standard CRUD operations like edit and delete. A modal window is open at the bottom, providing detailed information about the current user session, such as "Logged in as admin", "Authenticated Yes", and "Roles ROLE\_ADMIN + 1 more". The footer of the browser window shows performance metrics like 200 ms response time and 4.0 MB memory usage.

# 10. Symfony HttpClient

## Empêcher le spam avec une API

N'importe qui peut soumettre un commentaire, même des robots ou des spameurs. Nous pourrions ajouter un "captcha" au formulaire pour nous protéger des robots, ou nous pouvons utiliser des API tierces.

J'ai décidé d'utiliser le service gratuit [Akismet](#) pour montrer comment appeler une API et comment faire un appel "vers l'extérieur".

- ➔ Créez un compte gratuit sur [akismet.com](#) et récupérez la clé de l'API Akismet.

Au lieu d'utiliser une bibliothèque qui abstrait l'API d'Akismet, nous ferons directement tous les appels API avec Symfony/HttpClient. Faire nous-mêmes les appels HTTP est plus efficace (et nous permet de bénéficier de tous les outils de débogage de Symfony comme l'intégration avec le Symfony Profiler).

Nous allons concevoir une classe de vérification de spam.

- ➔ Créez une nouvelle classe dans `src/` nommée `SpamChecker` pour contenir la logique d'appel à l'API d'Akismet et l'interprétation de ses réponses
- ➔ Définissez la propriété `private readonly string $endpoint` pour stocker l'URL de l'API Akismet
- ➔ Définissez le constructeur pour injecter l'URL de l'API Akismet et le client HTTP

```
public function __construct()
{
    private readonly HttpClientInterface $client,
    #[Autowire('%env(string:AKISMET_KEY)%')]
    string $akismetKey
}
$this->endpoint = sprintf('https://%s.rest.akismet.com/1.1/comment-check', $akismetKey);
```

# Empêcher le spam avec une API

- ➔ Ajoutez une méthode `getSpamScore()` pour vérifier si un commentaire est du spam :

```
/**  
 * @return int Spam score: 0: not spam, 1: maybe spam, 2: blatant spam  
 */  
public function getSpamScore(Comment $comment, array $context): int  
{  
    $response = $this->client->request('POST', $this->endpoint, [  
        'body' => array_merge($context, [  
            'blog' => 'https://guestbook.example.com',  
            'comment_type' => 'comment',  
            'comment_author' => $comment->getAuthor(),  
            'comment_author_email' => $comment->getEmail(),  
            'comment_content' => $comment->getText(),  
            'comment_date_gmt' => $comment->getCreatedAt()->format('c'),  
            'blog_lang' => 'en',  
            'blog_charset' => 'UTF-8',  
            'is_test' => true,  
        ]),  
    ]);  
  
    $headers = $response->getHeaders();  
    if ('discard' === ($headers['x-akismet-pro-tip'][0] ?? '')) {  
        return 2;  
    }  
  
    $content = $response->getContent();  
    if (isset($headers['x-akismet-debug-help'][0])) {  
        throw new \RuntimeException(sprintf('Unable to check for spam: %s (%s).', $content, $headers['x-akismet-debug-help'][0]));  
    }  
  
    return 'true' === $content ? 1 : 0;  
}
```

Identifions ce que fait la méthode `getSpamScore()`.

Dans un premier temps, elle va transmettre à **Akismet** certaines données concernant le commentaire posté. Puis en fonction des données de la réponse, on retourne 3 valeurs différentes :

- `2` : si le commentaire est un "spam flagrant" ; On lèvera une exception pour le signaler.
- `1` : si le commentaire pourrait être du spam ;
- `0` : si le commentaire n'est pas du spam (ham).

# Identifier le spam dans les commentaires

Une façon simple de vérifier la présence de spam lorsqu'un nouveau commentaire est soumis est d'appeler le vérificateur de spam avant de stocker les données dans la base de données :

- ➔ Modifiez la méthode `show()` du contrôleur pour appeler le vérificateur de spam :

```
# [Autowire('%photo_dir%')] string $photoDir,  
+     SpamChecker $spamChecker,  
+ ): Response {  
+     $comment = new Comment();  
@@ ...  
+         $this->entityManager->persist($comment);  
+  
+         $context = [  
+             'user_ip' => $request->getClientIp(),  
+             'user_agent' => $request->headers->get('user-agent'),  
+             'referrer' => $request->headers->get('referer'),  
+             'permalink' => $request->getUri(),  
+         ];  
+         if (2 === $spamChecker->getSpamScore($comment, $context)) {  
+             throw new \RuntimeException('Blatant spam, go away!');  
+         }  
+  
+         $this->entityManager->flush();
```

- ➔ Essayez de soumettre un commentaire avec le mot "spam" dans le texte.
  - ✓ Le commentaire est stocké dans la base de données.
  - ⚡ Mais il n'est pas affiché sur la page de la conférence. Nous devons mettre à jour le contrôleur pour filtrer les commentaires marqués comme spam.

# Travaux pratique Kata

Créer une application Symfony pour gérer une bibliothèque de livres, avec des fonctionnalités d'authentification utilisateur et d'intégration avec l'API Google Books pour obtenir les informations complètes des livres à partir du titre et de l'auteur.

## Fonctionnalités Principales

### 1. Gestion des Livres

- Données d'un livre : titre, auteur, date de publication, ISBN, date d'ajout, de modification et l'utilisateur qui a ajouté le livre.
- Lister les Livres : Afficher une liste de tous les livres dans la bibliothèque.
- Voir les Détails d'un Livre : Afficher les détails d'un livre spécifique.
- Ajouter un Livre (authentification requise) : Permettre aux utilisateurs d'ajouter de nouveaux livres en entrant uniquement le titre et l'auteur. Les autres informations seront récupérées via l'API Google Books.
- Modifier un Livre (authentification requise) : Permettre aux utilisateurs de modifier les informations d'un livre existant.
- Supprimer un Livre (authentification requise) : Permettre aux utilisateurs de supprimer un livre de la bibliothèque.

### 2. Authentification Utilisateur

- Inscription : Les nouveaux utilisateurs peuvent créer un compte.
- Connexion : Les utilisateurs existants peuvent se connecter.
- Déconnexion : Les utilisateurs peuvent se déconnecter.
- Sécurité : Restreindre l'accès aux fonctionnalités de gestion des livres (ajout, modification, suppression) aux utilisateurs authentifiés uniquement.

### 3. Intégration avec l'API Google Books ([https://developers.google.com/books/docs/v1/getting\\_started?hl=fr](https://developers.google.com/books/docs/v1/getting_started?hl=fr))

- Obtenir les Informations Complètes des Livres : Utiliser l'API Google Books pour récupérer les informations complètes d'un livre à partir du titre et de l'auteur fournis par l'utilisateur.
- Affichage des Informations : Afficher les informations récupérées dans la fiche du livre.

# 11. Les tests

# Installer PHPUnit

Comme nous commençons à ajouter, de plus en plus de fonctionnalités dans l'application, c'est probablement le bon moment pour parler des tests.

Symfony s'appuie sur PHPUnit pour les tests unitaires.

- ▶ **Installons-le :**

```
symfony composer req phpunit/phpunit:^11 -W --dev
```

- ▶ **Modifier le contenu du fichier `phpunit.xml.dist` pour ajouter la configuration de PHPUnit 11 :**

```
<?xml version="1.0" encoding="UTF-8"?>

<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="vendor/phpunit/phpunit/phpunit.xsd"
  backupGlobals="false" colors="true"
  bootstrap="tests/bootstrap.php"
  failOnRisky="true" failOnWarning="true"
  failOnEmptyTestSuite="true"
  beStrictAboutChangesToGlobalState="true"
  beStrictAboutOutputDuringTests="true"
  cacheDirectory=".phpunit.cache"
>
  <php>
    <ini name="display_errors" value="1" />
    <ini name="error_reporting" value="-1" />
    <server name="APP_ENV" value="test" force="true" />
    <server name="SHELL_VERBOSITY" value="-1" />
    <server name="SYMFONY_PHPUNIT_REMOVE" value="" />
    <server name="SYMFONY_PHPUNIT_VERSION" value="11" />
    <server name="SYMFONY_DEPRECATED_HELPER" value="disabled" />
  </php>
  <testsuites>
    <testsuite name="Project Test Suite">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
  <source>
    <include>
      <directory>src</directory>
    </include>
  </source>
</phpunit>
```

## Générer une classe de test unitaire

`SpamChecker` est la première classe pour laquelle nous allons écrire des tests.

- ➔ Générez un test unitaire avec le maker :

```
symfony console make:test TestCase SpamCheckerTest
```

Suite à cette commande, une nouvelle classe `SpamCheckerTest` est créée dans le répertoire `tests/`. Elle contient une méthode de test `testSomething` qui vérifie que `true` est égal à `true`.

Tester le `SpamChecker` est un défi, car nous ne voulons certainement pas utiliser l'API Akismet. Nous allons mocker l'API.

# Écrire des tests unitaires

- ▶ Supprimez la méthode `testSomething()` et écrivons un premier test pour le cas où l'API renverrait une erreur :

```
class SpamCheckerTest extends TestCase
{
    public function testSpamScoreWithInvalidRequest(): void
    {
        ($comment = new Comment())->setCreatedAtValue();

        $client = new MockHttpClient([new MockResponse('invalid', ['response_headers' => ['x-akismet-debug-help: Invalid key']])));
        $checker = new SpamChecker($client, 'abcde');

        static::expectException(\RuntimeException::class);
        static::expectExceptionMessage('Unable to check for spam: invalid (Invalid key).');
        $checker->getSpamScore(comment: $comment, context: []);
    }
}
```

La classe `MockHttpClient` permet de simuler n'importe quel serveur HTTP. Elle prend un tableau d'instances `MockResponse` contenant le corps et les en-têtes de réponse attendus.

Elle implémente l'interface `HttpClientInterface` dont a besoin notre `SpamChecker`.

Ensuite, nous appelons la méthode `getSpamScore()` et vérifions qu'une exception est levée via la méthode `expectException()` de PHPUnit.

- ▶ Lancez notre premier test unitaire :

```
symfony php bin/phpunit
```

# Ecrire le test pour le cas où l'API renverrai un spam flagrant

- ▶ Écrivons un autre test pour le cas où l'API renverrait un spam flagrant :

```
#[DataProvider('provideComments')]
public function testSpamScore(int $expectedScore, ResponseInterface $response, Comment $comment, array $context)
{
    $client = new MockHttpClient([$response]);
    $checker = new SpamChecker($client, 'abcde');

    $score = $checker->getSpamScore($comment, $context);
    static::assertSame($expectedScore, $score);
}
```

- ▶ Ajoutez une méthode `provideComments()` pour fournir des données de test :

```
/** @return iterable<string, array<mixed>> */
public static function provideComments(): iterable
{
    ($comment = new Comment())->setCreatedAtValue();
    $response = new MockResponse('', ['response_headers' => ['x-akismet-pro-tip: discard']]);
    yield 'blatant_spam' => [2, $response, $comment, []];
    $response = new MockResponse('true');
    yield 'spam' => [1, $response, $comment, []];
    $response = new MockResponse('false');
    yield 'ham' => [0, $response, $comment, []];
}
```

Les `DataProvider` de PHPUnit nous permettent de réutiliser la même logique de test pour plusieurs scénarios.

- ▶ Lancez les tests unitaires :

```
symfony php bin/phpunit
```

## Écrire des tests fonctionnels pour les contrôleurs

Tester les contrôleurs est un peu différent de tester une classe PHP "ordinaire" car nous voulons les exécuter dans le contexte d'une requête HTTP. On appelle ces tests des tests fonctionnels.

Le maker de Symfony peut générer des classes de test fonctionnel pour les contrôleurs.

- ➔ Générez une classe de test fonctionnel :

```
symfony console make:test WebTestCase ConferenceControllerTest
```

Suite à cette commande, une nouvelle classe `ConferenceControllerTest` est créée dans le répertoire `tests/`.

- ➔ Supprimez la méthode `testSomething()` et écrivez un test pour vérifier que la page d'accueil fonctionne :

```
public function testIndex(): void
{
    $client = static::createClient();
    $crawler = $client->request('GET', '/');

    static::assertResponseIsSuccessful();
    static::assertSelectorTextContains('h2', 'Give your feedback!');
}
```

Utiliser `Symfony\Bundle\FrameworkBundle\Test\WebTestCase` à la place de `PHPUnit\Framework\TestCase` comme classe de base pour nos tests nous fournit une abstraction bien pratique pour les tests fonctionnels.

La variable `$client` simule un navigateur. Au lieu de faire des appels HTTP au serveur, il appelle directement l'application Symfony. Cette stratégie présente plusieurs avantages ✓ :

- Elle est beaucoup plus rapide que les allers-retours entre le client et le serveur,
- Mais elle permet aussi aux tests d'analyser l'état des services après chaque requête HTTP.

Ce premier test vérifie que la page d'accueil renvoie une réponse HTTP 200 ✓

# Configurer l'environnement de test

Par défaut, les tests PHPUnit sont exécutés dans l'environnement Symfony test tel qu'il est défini dans le fichier de configuration de PHPUnit `phpunit.xml.dist` :

```
<phpunit>
  <php>
    <ini name="display_errors" value="1" />
    <ini name="error_reporting" value="-1" />
    <server name="APP_ENV" value="test" force="true" />
    <server name="SHELL_VERBOSITY" value="-1" />
    <server name="SYMFONY_PHPUNIT_REMOVE" value="" />
    <server name="SYMFONY_PHPUNIT_VERSION" value="11" />
    <server name="SYMFONY_DEPRECATED_HELPER" value="disabled" />
  </php>
</phpunit>
```

- ▶ Pour faire fonctionner les tests, nous devons définir la clé secrète `AKISMET_KEY` pour cet environnement test dans le fichier `.env.test.local` :

```
# .env.test.local
AKISMET_KEY=abcde
```

- ▶ Mettez à jour le fichier `test/bootstrap.php` pour supporter `phpunit 11` :

```
<?php

use Symfony\Component\Dotenv\Dotenv;
+use Symfony\Component\ErrorHandler\ErrorHandler;

require dirname(__DIR__).'/vendor/autoload.php';
+set_exception_handler([new ErrorHandler(), 'handleException']);
```

## Utiliser une base de données de test

Comme nous l'avons déjà vu, la commande Symfony définit automatiquement la variable d'environnement `DATABASE_URL`. Quand `APP_ENV` a pour valeur `test`, comme c'est le cas lors de l'exécution de PHPUnit, cela change le nom de la base de données de `app` en `app_test` pour que les tests utilisent leur propre base de données :

```
# config/packages/doctrine.yaml
when@test:
    doctrine:
        dbal:
            # "TEST_TOKEN" is typically set by ParaTest
            dbname_suffix: '_test%env(default::TEST_TOKEN)%'
```

Cela est très important, car nous aurons besoin d'un jeu de données stable pour exécuter nos tests et nous ne voulons certainement pas écraser celui stocké dans la base de développement.

- ➔ Avant de pouvoir lancer les tests, nous devons "initialiser" la base de données test (créez la base de données et jouez les migrations) :

```
APP_ENV=test symfony console doctrine:database:create  
APP_ENV=test symfony console doctrine:migrations:migrate -n
```

Si vous lancez les tests maintenant, PHPUnit n'interagira plus avec votre base de données de développement. Pour lancer les nouveaux tests uniquement, passez le chemin de leur classe en argument.

- ➔ Lancez les tests unitaires :

```
symfony php bin/phpunit tests/ConferenceControllerTest.php
```

## Définir des fixtures (données de test)

Pour pouvoir tester la liste des commentaires, la pagination et la soumission du formulaire, nous devons remplir la base de données avec quelques données. Nous voulons également que les données soient identiques entre les cycles de tests pour qu'ils réussissent. Les fixtures sont exactement ce dont nous avons besoin.

-  **Installez le composant `doctrine/doctrine-fixtures-bundle`** :

```
symfony composer req --dev doctrine/doctrine-fixtures-bundle
```

Un nouveau répertoire `src/DataFixtures/` a été créé lors de l'installation, avec une classe d'exemple prête à être personnalisée.

Dans le fichier `src/DataFixtures/AppFixtures.php`, nous allons ajouter des conférences et des commentaires pour les tests.

- ➔ Remplacez le contenu de la méthode `load()` par le code suivant qui ajoute deux conférences et un commentaire :

```
public function load(ObjectManager $manager): void
{
    $amsterdam = new Conference();
    $amsterdam->setCity('Amsterdam');
    $amsterdam->setYear('2019');
    $amsterdam->setIsInternational(true);
    $manager->persist($amsterdam);

    $paris = new Conference();
    $paris->setCity('Paris');
    $paris->setYear('2020');
    $paris->setIsInternational(false);
    $manager->persist($paris);

    $comment1 = new Comment();
    $comment1->setConference($amsterdam);
    $comment1->setAuthor('Fabien');
    $comment1->setEmail('fabien@example.com');
    $comment1->setText('This was a great conference.');
    $manager->persist($comment1);
    $manager->flush();
}
```

# Charger les fixtures

Lorsque nous chargerons les données de test, toutes les données présentes seront supprimées, y compris celles de l'admin.

- ▶ Pour éviter cela, modifions les fixtures :

```
class AppFixtures extends Fixture
{
    public function __construct(
        private PasswordHasherFactoryInterface $passwordHasherFactory,
    ) {
    }

    public function load(ObjectManager $manager): void
    {

        ...

        $manager->persist($comment1);

        $admin = new Admin();
        $admin->setRoles(['ROLE_ADMIN']);
        $admin->setUsername('admin');
        $admin->setPassword($this->passwordHasherFactory->getPasswordHasher(Admin::class)->hash('admin'));
        $manager->persist($admin);
        $manager->flush();
    }
}
```

- ▶ Chargez les données de test pour l'environnement/la base de données de test :

```
APP_ENV=test symfony console doctrine:fixtures:load --no-interaction
```

# Parcourir un site web avec des tests fonctionnels

Comme nous l'avons vu, le client `HTTP` utilisé dans les tests simule un navigateur, afin que nous puissions parcourir le site comme si nous utilisions un navigateur.

- ▶ Ajoutez à la classe `ConferenceControllerTest` un nouveau test qui clique sur une page de conférence depuis la page d'accueil :

```
public function testConferencePage() : void
{
    $client = static::createClient();
    $crawler = $client->request('GET', '/');
    static::assertCount(2, $crawler->filter('h4'));

    $client->clickLink('View');
    static::assertPageTitleContains('Amsterdam');

    static::assertResponseIsSuccessful();
    static::assertSelectorTextContains('h2', 'Amsterdam 2019');

    static::assertSelectorExists('div:contains("There are 1 comments")');
}
```

- ▶ Exécutez les tests :

```
symfony php bin/phpunit tests/ConferenceControllerTest.php
```

## Décrivons ce qu'il se passe dans ce test :

- Comme pour le premier test, nous allons sur la page d'accueil
- La méthode `request()` retourne une instance de `Crawler` qui aide à trouver des éléments sur la page (comme des liens, des formulaires, ou tout ce que vous pouvez atteindre avec des sélecteurs CSS ou XPath) ;
- Grâce à un sélecteur CSS, nous testons que nous avons bien deux conférences listées sur la page d'accueil ;
- On clique ensuite sur le lien "View" (comme il n'est pas possible de cliquer sur plus d'un lien à la fois, Symfony choisit automatiquement le premier qu'il trouve) ;
- Nous vérifions le titre de la page, la réponse et le `<h2>` de la page pour être sûr d'être sur la bonne page (nous aurions aussi pu vérifier la route correspondante) ;
- Enfin, nous vérifions qu'il y a 1 commentaire sur la page. `div:contains()` n'est pas un sélecteur CSS valide, mais Symfony a quelques ajouts intéressants, empruntés à jQuery.

# Soumettre un formulaire dans un test fonctionnel

 Voulez-vous passer au niveau supérieur ?

Essayez d'ajouter un nouveau commentaire avec une photo sur une conférence, à partir d'un test, en simulant une soumission de formulaire. Cela semble ambitieux, n'est-ce pas ?

-  Regardez le code nécessaire : pas plus compliqué que ce que nous avons déjà écrit. Ajoutez `cd test` à notre classe `ConferenceControllerTest`

```
public function testCommentSubmission() : void
{
    $client = static::createClient();
    $client->request('GET', '/conference/amsterdam-2019');

    $client->submitForm('Submit', [
        'comment_form[author]' => 'Fabien',
        'comment_form[text]' => 'Some feedback from an automated functional test',
        'comment_form[email]' => 'me@automat.ed',
        'comment_form[photo]' => dirname(__DIR__, 2).'./public/images/under-construction.gif',
    ]);

    static::assertResponseRedirects();
    $client->followRedirect();
    static::assertSelectorExists('div:contains("There are 2 comments")');
}
```

Pour soumettre un formulaire via `submitForm()`, recherchez les noms de champs grâce aux outils de développement du navigateur ou via l'onglet **Form** du **Symfony Profiler**. Notez la réutilisation pratique de l'image en construction !

- ➔ Relancez les tests pour vérifier que tout est bon :

```
symfony php bin/phpunit tests/Controller/ConferenceControllerTest.php
```



- ➔ Si vous voulez vérifier le résultat dans un navigateur, arrêtez le serveur web et relancer le pour l'environnement test :

```
symfony server:stop  
APP_ENV=test symfony server:start
```

## Recharger les données de test

- ▶ Si vous effectuez les tests une deuxième fois, ils devraient échouer.



⚠️ Comme il y a maintenant plus de commentaires dans la base de données, l'assertion qui vérifie le nombre de commentaires est erronée.

- ▶ Nous devons réinitialiser l'état de la base de données entre chaque exécution, en rechargeant les données de test avant chacune d'elles :

```
APP_ENV=test symfony console doctrine:fixtures:load -n  
symfony php bin/phpunit tests/ConferenceControllerTest.php
```

Yeap ! Les tests passent à nouveau.



## Automatiser votre workflow avec un Makefile

🤔 Il est assez pénible d'avoir à se souvenir d'une séquence de commandes pour exécuter les tests. Cela devrait au moins être documenté, même si cette documentation ne devrait être consultée qu'en dernier recours.

👉 Et si on automatisait plutôt les opérations récurrentes ?

Cela servirait aussi de documentation rapidement accessible aux autres, et rendrait le développement plus facile et plus productif.



- ➡ L'utilisation d'un fichier **Makefile** est une façon d'automatiser les commandes :

```
# Makefile
SHELL := /bin/bash

tests:
    APP_ENV=test symfony console doctrine:database:drop --force || true
    APP_ENV=test symfony console doctrine:database:create
    APP_ENV=test symfony console doctrine:schema:update --force
    APP_ENV=test symfony console doctrine:fixtures:load -n
    APP_ENV=dev symfony php bin/phpunit ${MAKECMDGOALS}

.PHONY: tests
```

⚠ Dans une règle **Makefile**, l'indentation doit être une seule tabulation et non des espaces.

Notez l'option **-n** sur la commande Doctrine ; c'est une option standard sur les commandes Symfony qui les rend non interactives.

- ➡ Chaque fois que vous voulez exécuter les tests, utilisez **make tests** :

```
make tests
```

## Réinitialiser la base de données après chaque test

Réinitialiser la base de données après chaque test, c'est bien ! Mais avoir des tests vraiment indépendants, c'est encore mieux. 🚀🚀

Nous ne voulons pas qu'un test s'appuie sur les résultats des précédents. Le changement de l'ordre des tests ne devrait pas changer le résultat. Comme nous allons le découvrir maintenant, ce n'est pas le cas pour le moment.

- ➔ Déplacez le test `testConferencePage` après `testCommentSubmission` et rejouez les tests avec `make tests`
  - 🔴 Les tests échouent maintenant. Le test `testCommentSubmission` ne peut pas trouver la conférence car elle a été supprimée par le test `testConferencePage`.

Pour résoudre ce problème, nous devons réinitialiser la base de données après chaque test en installant le composant `DoctrineTestBundle`

- ➔ Installez le composant `DoctrineTestBundle` :

```
symfony composer require "dama/doctrine-test-bundle:^8" --dev
```

Vous devrez confirmer l'application de la recette (car il ne s'agit pas d'un bundle "officiellement" supporté) :

La recette `dama/doctrine-test-bundle` a mis à jour le fichier `phpunit.xml.dist` pour ajouter une configuration spécifique à `DoctrineTestBundle`. Celle-ci n'est pas compatible avec la configuration de PHPUnit 11 que nous avons ajoutée précédemment.

- ➔ Modifiez le fichier `phpunit.xml.dist` pour ajouter la configuration de `DoctrineTestBundle` :

```
<extensions>
    <bootstrap class="DAMA\DoctrineTestBundle\PHPUnit\PHPUnitExtension" />
</extensions>
```

Et voilà. Toute modification apportée pendant les tests est automatiquement annulée à la fin de chaque test.

- ➔ Les tests devraient passer à nouveau :

```
make tests
```

# Choisir le bon type de test

Nous avons créé trois types de tests jusqu'à maintenant. Bien que nous n'ayons utilisé le bundle maker que pour générer des tests unitaires, nous aurions tout aussi bien pu l'utiliser pour générer les classes des autres tests :

```
symfony console make:test WebTestCase Controller\\ConferenceController  
symfony console make:test PantherTestCase Controller\\ConferenceController
```

Le bundle maker supporte la génération des types de tests suivants en fonction de la manière dont vous voulez tester votre application :

- `TestCase` : Tests PHPUnit basiques ;
- `KernelTestCase` : Tests basiques ayant accès aux services Symfony ;
- `WebTestCase` : Pour exécuter des scénarios à la manière d'un navigateur, mais sans exécution du code JavaScript ;
- `ApiTestCase` : Pour jouer des scénarios orientés API ;
- `PantherTestCase` : Pour jouer des scénarios e2e, en utilisant un vrai navigateur ou client HTTP et un vrai serveur web.

Les tests sont importants pour garantir que votre application fonctionne correctement. Ils vous permettent de vérifier que les nouvelles fonctionnalités que vous ajoutez ne cassent pas les fonctionnalités existantes.

- ➔ **Commitez les modifications :**

```
git add .  
git commit -m "Add tests"
```

# (Bonus) Les outils pour la qualité de code

# PHPStan

Le composant `phpunit` est un outil de test unitaire. Il existe d'autres outils pour tester la qualité de votre code. Tel que `phpstan` qui est un outil d'analyse statique du code PHP et `phpcs-fixer` qui est un outil de correction de code.

PHPStan est un outil d'analyse statique du code PHP qui vous aide à détecter les erreurs potentielles dans votre code. Il est capable de détecter les erreurs de type, les erreurs de logique, les erreurs de syntaxe, les erreurs de performance et les erreurs de style.

- ➔ Installez le composant `phpstan` :

```
symfony composer req --dev phpstan/phpstan
```

- ➔ Ajouter à notre fichier Makefile la commande pour lancer `phpstan` :

```
phpstan:  
APP_ENV=dev symfony php vendor/bin/phpstan analyse --level max
```

- ➔ Lancez `phpstan` :

```
make phpstan
```

⚠ Vous remarquerez que PHPStan a détecté des erreurs dans notre code. C'est une bonne chose, car cela signifie que nous pouvons les corriger avant qu'elles ne deviennent des problèmes.

# PHPCs-fixer

PHP-CS-Fixer est un outil de correction de code qui vous aide à maintenir un code propre et conforme à un ensemble de règles de style. Il peut corriger automatiquement les erreurs de style de votre code.

- ➔ Installez le composant `php-cs-fixer` :

```
symfony composer req --dev friendsofphp/php-cs-fixer
```

- ➔ Ajouter à notre fichier Makefile la commande pour lancer `php-cs-fixer` :

```
php-cs-fixer:  
    APP_ENV=dev symfony php vendor/bin/php-cs-fixer fix  
  
php-cs-fixer-dry-run:  
    APP_ENV=dev symfony php vendor/bin/php-cs-fixer fix --dry-run
```

- ➔ Lancer la correction du code :

```
make php-cs-fixer
```

Le Cs-fixer corrige les erreurs de style dans notre code. Cela nous permet de nous concentrer sur l'écriture de code et de ne pas perdre de temps à le formater.

Pour automatiser la vérification de la qualité de notre code, nous pouvons ajouter une commande `quality` à notre `Makefile` qui exécute `phpstan`, `php-cs-fixer` et les tests.

- ➔ Ajouter la commande `quality` à notre `Makefile` :

```
quality: phpstan php-cs-fixer tests
```

- ➔ Lancer la vérification de la qualité du code :

```
make quality
```

Grâce à cette commande, nous pouvons vérifier la qualité de notre code en une seule commande. Cela nous permet de nous permet de gagner du temps et de nous concentrer sur l'écriture de code.

- ➔ Commitez les modifications :

```
git add .
git commit -m "Add quality tools"
```

# 12. Faire de l'asynchrone

## Marquer les commentaires

Vérifier la présence de spam pendant le traitement de la soumission du formulaire peut entraîner certains problèmes.

🤔 Si l'API d'Akismet devient lente, notre site web sera également lent pour les internautes. Mais pire encore 😱, si nous atteignons le délai d'attente maximal ou si l'API d'Akismet n'est pas disponible, nous pourrions perdre des commentaires.

Idéalement, nous devrions stocker les données soumises, sans les publier, et renvoyer une réponse immédiatement. La vérification du spam pourra être faite par la suite.

Nous avons besoin d'introduire un état (`state`) pour les commentaires : `submitted`, `spam` et `published`.

- ➔ Ajoutez la propriété `state` à la classe `Comment` :

```
symfony console make:entity Comment
```

- ➔ Ajouter une classe enum `App\Entity\Enum\CommentStateEnum.php` pour définir les états possible :

```
namespace App\Entity\Enum;

enum CommentStateEnum : string
{
    case Submitted = 'submitted';
    case Spam = 'spam';
    case Published = 'published';
}
```

Au niveau de notre entité `Comment`, utilisons cette classe enum pour définir les états possibles. Et nous devrions également nous assurer que, par défaut, le paramètre `state` est initialisé avec la valeur `submitted`

- ➔ Mettez à jour la classe `Comment` :

```
- #[ORM\Column(length: 255)]
- private ?string $state = null;
+ #[ORM\Column(length: 255, options: ['default' => CommentStateEnum::Submitted->value])]
+ private ?CommentStateEnum $state = CommentStateEnum::Submitted;
...
- public function getState(): ?string
+ public function getState(): ?CommentStateEnum
...
- public function setState(?string $state): static
+ public function setState(?CommentStateEnum $state): static
```

# Migration de la base de données

- ➔ Créez une migration de base de données :

```
symfony console make:migration --formatted
```

- ➔ Modifiez la migration pour mettre à jour tous les commentaires existants comme étant published par défaut :

```
    $this->addSql('ALTER TABLE comment ADD state VARCHAR(255) DEFAULT \'submitted\' NOT NULL');
+      $this->addSql("UPDATE comment SET state='published'");
}
```

- ➔ Exécutez la migration :

```
symfony console doctrine:migrations:migrate -n
```

# Mis à jour de notre code

- ➔ Modifiez la logique d'affichage pour éviter que des commentaires non publiés n'apparaissent sur le site :

```
# src/Repository/CommentRepository.php

    $query = $this->createQueryBuilder('c')
        ->andWhere('c.conference = :conference')
+        ->andWhere('c.state = :state')
+        ->setParameter('conference', $conference)
+        ->setParameter('state', CommentStateEnum::Published)
```

- ➔ Modifiez la configuration d'EasyAdmin pour voir l'état du commentaire :

```
# src/Controller/Admin/CommentCrudController.php
    ->setLabel('Photo')
    ->onlyOnIndex()
;
+    yield TextField::new('state');
```

- ➔ N'oubliez pas de modifier les tests en renseignant le state dans les fixtures `AppFixtures`, profitons aussi pour ajouter un autre commentaire :

```
$comment1->setText('This was a great conference.');
+    $comment1->setState(CommentStateEnum::Published);
$manager->persist($comment1);

+    $comment2 = new Comment();
+    $comment2->setConference($amsterdam);
+    $comment2->setAuthor('Lucas');
+    $comment2->setEmail('lucas@example.com');
+    $comment2->setText('I think this one is going to be moderated.');
+    $comment2->setState(CommentStateEnum::Published);
+    $manager->persist($comment2);
```

# Mis à jour de notre code

- ▶ Pour les tests du contrôleur `tests/Controller/ConferenceControllerTest.php`, simulez la validation :

```
+use App\Repository\CommentRepository;
+use Doctrine\ORM\EntityManagerInterface;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class ConferenceControllerTest extends WebTestCase
@@ -22,10 +24,16 @@ class ConferenceControllerTest extends WebTestCase
    $client->submitForm('Submit', [
        'comment_form[author]' => 'Fabien',
        'comment_form[text]' => 'Some feedback from an automated functional test',
-       'comment_form[email]' => 'me@automat.ed',
+       'comment_form[email]' => $email = 'me@automat.ed',
        'comment_form[photo]' => dirname(__DIR__, 2).'./public/images/under-construction.gif',
    ]);
    static::assertResponseRedirects();
+
+   // simulate comment validation
+   $comment = static::getContainer()->get(CommentRepository::class)->findOneByEmail($email);
+   $comment->setState(CommentStateEnum::Published);
+   static::getContainer()->get(EntityManagerInterface::class)->flush();

    $client->followRedirect();
-   static::assertSelectorExists('div:contains("There are 2 comments")');
+   self::assertSelectorExists('div:contains("There are 3 comments")');
```

À partir d'un test PHPUnit, vous pouvez obtenir n'importe quel service depuis le conteneur grâce à `static::getContainer()->get('service');` il donne également accès aux services non publics.

- ▶ Lancez les tests :

```
make tests
```

# Comprendre Messenger

La gestion du code asynchrone avec Symfony est faite par le composant Messenger :

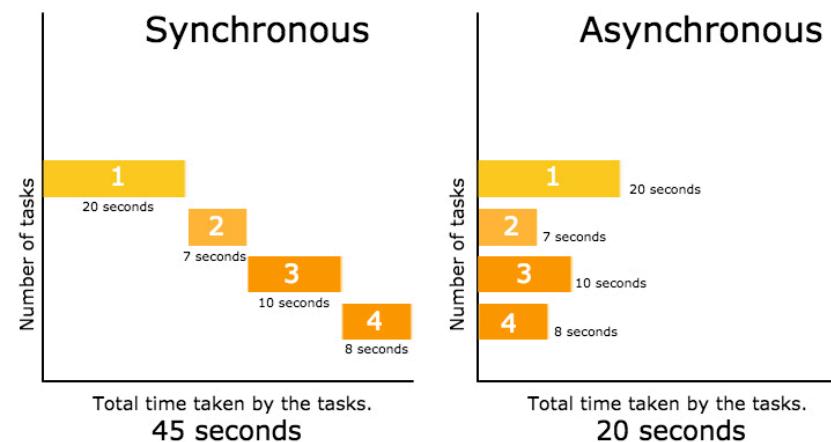
- ➔ Installez le composant Messenger :

```
symfony composer require doctrine-messenger
```

Lorsqu'une action doit être exécutée de manière asynchrone, envoyez un message à **un messenger bus**. Le bus stocke le message dans une file d'attente et rend immédiatement la main pour permettre au flux des opérations de reprendre aussi vite que possible.

Un **consumer** s'exécute continuellement en arrière-plan pour lire les nouveaux messages dans la file d'attente et exécuter la logique associée. Le consumer peut s'exécuter sur le même serveur que l'application web, ou sur un serveur séparé.

C'est très similaire à la façon dont les requêtes HTTP sont traitées, **sauf que nous n'avons pas de réponse**.



# Coder un gestionnaire de messages

Un message est une classe de données (data object), qui ne doit contenir aucune logique. Il sera sérialisé pour être stocké dans une file d'attente, donc ne stockez que des données "simples" et sérialisables.

- ➔ Générez nos classe de Message et Handler via le maker :

```
symfony console make:message CommentMessage
```

Une classe `CommentMessage` est générée dans le répertoire `src/Message` et une classe `CommentMessageHandler` est générée dans le répertoire `src/MessageHandler`.

- ➔ Modifiez la classe `CommentMessage` pour ajouter un contexte :

```
namespace App\Message;

final readonly class CommentMessage
{
    public function __construct(
        public int $id,
        public array $context = [],
    ) { }
}
```

Dans le monde de Messenger, nous n'avons pas de contrôleurs, mais des gestionnaires de messages.

- ▶ Sous un nouveau namespace `App\MessageHandler`, modifiez la classe `CommentMessageHandler` qui saura comment gérer les messages `CommentMessage` :

```
namespace App\MessageHandler;

use App\Message\CommentMessage;
use App\Repository\CommentRepository;
use App\SpamChecker;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\Messenger\Attribute\AsMessageHandler;

#[AsMessageHandler]
final readonly class CommentMessageHandler
{
    public function __construct(
        private EntityManagerInterface $entityManager,
        private SpamChecker $spamChecker,
        private CommentRepository $commentRepository,
    ) {}

    public function __invoke(CommentMessage $message) : void
    {
        $comment = $this->commentRepository->find($message->id);
        if (null === $comment) {
            return;
        }

        $comment->setState(CommentStateEnum::Published);
        if (2 === $this->spamChecker->getSpamScore($comment, $message->context)) {
            $comment->setState(CommentStateEnum::Spam);
        }

        $this->entityManager->flush();
    }
}
```

`#[AsMessageHandler]` aide Symfony à enregistrer et à configurer automatiquement la classe en tant que gestionnaire Messenger.

Par convention, la logique d'un gestionnaire réside dans une méthode appelée `__invoke()`. Le type `CommentMessage` précisé en tant qu'argument unique de cette méthode indique à Messenger quelle classe elle va gérer.

- ➔ Modifiez le contrôleur `src/Controller/ConferenceController.php` pour utiliser le nouveau système :

```
use App\Repository\ConferenceRepository;
+use App\Message\CommentMessage;
- use App\SpamChecker;
+use Symfony\Component\Messenger\MessageBusInterface;

@@ ...
    CommentRepository $commentRepository,
-    SpamChecker $spamChecker,
+    MessageBusInterface $bus,
    #[Autowire('%photo_dir%')] string $photoDir,
): Response {
@@ ...

    $this->entityManager->persist($comment);
+    $this->entityManager->flush();
@@ ...

-    if (2 === $spamChecker->getSpamScore($comment, $context)) {
-        throw new \RuntimeException('Blatant spam, go away!');
-    }
-
-    $this->entityManager->flush();
+    $bus->dispatch(new CommentMessage($comment->getId(), $context));
```

Au lieu de dépendre du `SpamChecker`, nous envoyons maintenant un message dans le **bus**. Le gestionnaire décide alors ce qu'il en fait.

Nous avons fait quelque chose que nous n'avions pas prévu. Nous avons découplé notre contrôleur du vérificateur de spam, et déplacé la logique vers une nouvelle classe, le gestionnaire. C'est un cas d'utilisation parfait pour le bus.

- ➔ **Testez le code, il fonctionne.**

Tout se fait encore de manière synchrone, mais le code est probablement déjà "meilleur".

## Faire vraiment de l'asynchrone

Par défaut, les gestionnaires sont appelés de manière synchrone. Pour les rendre asynchrones, vous devez :

- ➔ configurer explicitement la file d'attente à utiliser pour chaque gestionnaire dans le fichier de configuration `config/packages/messenger.yaml` :

```
Symfony\Component\Notifier\Message\SmsMessage: async  
  # Route your messages to the transports  
  -  
  +  # 'App\Message\YourMessage': async  
  +  App\Message\CommentMessage: async
```

La configuration indique au bus d'envoyer les instances de `App\Message\CommentMessage` à la file d'attente `async`, qui est définie par un DSN (`MESSENGER_TRANSPORT_DSN`), qui pointe vers Doctrine tel que défini dans le fichier `.env`. En clair, nous utilisons PostgreSQL comme file d'attente pour nos messages.

💡 En coulisses, Symfony utilise le système pub/sub intégré, performant, dimensionnable (`LISTEN/NOTIFY`). Vous pouvez aussi lire le chapitre sur RabbitMQ si vous voulez l'utiliser à la place de PostgreSQL comme gestionnaire de messages.

## Consommer des messages

Si vous essayez de soumettre un nouveau commentaire, le vérificateur de spam ne sera plus appelé. Ajoutez un appel à la fonction `error_log()` dans la méthode `getSpamScore()` pour le confirmer. Au lieu d'avoir un nouveau commentaire, un message est en attente dans la file d'attente, prêt à être consommé par d'autres processus.

Comme vous pouvez l'imaginer, Symfony est livré avec une commande pour consommer les messages.

-  Exécutez-la maintenant :

```
symfony console messenger:consume async -vv
```

Cette commande devrait immédiatement consommer le message envoyé pour le commentaire soumis :

```
[OK] Consuming messages from transports "async".  
// The worker will automatically exit once it has received a stop signal via the messenger:stop-workers command.  
// Quit the worker with CONTROL-C.  
  
11:30:20 INFO [messenger] Received message App\Message\CommentMessage ["message" => App\Message\CommentMessage^ { ... }, "class" => "App\Message\CommentMessage"]  
11:30:20 INFO [http_client] Request: "POST https://80cea32be1f6.rest.akismet.com/1.1/comment-check"  
11:30:20 INFO [http_client] Response: "200 https://80cea32be1f6.rest.akismet.com/1.1/comment-check"  
11:30:20 INFO [messenger] Message App\Message\CommentMessage handled by App\MessageHandler\CommentMessageHandler::__invoke ["message" => App\Message\CommentMessage]  
11:30:20 INFO [messenger] App\Message\CommentMessage was handled successfully (acknowledging to transport). ["message" => App\Message\CommentMessage^]
```

L'activité du **consumer** de messages est enregistrée dans les logs, mais vous pouvez avoir un affichage instantané dans la console en passant l'option **-vv**. Vous devriez même voir l'appel vers l'API d'Akismet.

- ➔ Pour arrêter le consumer, appuyez sur **Ctrl+C**.

## Lancer des workers en arrière-plan

Au lieu de lancer le consumer à chaque fois que nous publions un commentaire et de l'arrêter immédiatement après, nous voulons l'exécuter en continu sans avoir trop de fenêtres ou d'onglets du terminal ouverts.

La commande `symfony` peut gérer des commandes en tâche de fond ou des workers en utilisant l'option `daemon` (`-d`) sur la commande `run`.

- ➔ Exécutez à nouveau le consumer du message, mais en tâche de fond :

```
symfony run -d --watch=config,src,templates,vendor symfony console messenger:consume async
```

💡 L'option `--watch` indique à Symfony que la commande doit être redémarrée chaque fois qu'il y a un changement dans un des fichiers des répertoires `config/`, `vendor/`, `src/` ou `templates/`.

Si le consumer cesse de fonctionner pour une raison quelconque (limite de mémoire, bogue, etc.), il sera redémarré automatiquement. Et s'il tombe en panne trop rapidement, la commande `symfony` s'arrêtera.

Les logs sont diffusés en continu par la commande `symfony server:log`, en même temps que ceux de PHP, du serveur web et de l'application.

- ➔ Utilisez la commande `server:status` pour lister tous les workers en arrière-plan gérés pour le projet en cours :

```
symfony server:status
Web server listening on https://127.0.0.1:8000
  Command symfony console messenger:consume async running with PID 15774 (watching config/, src/, templates/)
```

- ➔ Pour arrêter un worker, arrêtez le serveur web ou tuez le PID (identifiant du processus) donné par la commande `server:status` :

```
kill 15774
```

## Renvoyer des messages ayant échoué

Que faire si Akismet est en panne alors qu'un message est en train d'être consommé ? Il n'y a aucun impact pour les personnes qui soumettent des commentaires, mais le message est perdu et le spam n'est pas vérifié.

Messenger dispose d'un mécanisme de relance lorsqu'une exception se produit lors du traitement d'un message, configuré sur le transport `async` dans le fichier `config/packages/messenger.yaml` :

```
framework:
  messenger:
    failure_transport: failed

  transports:
    # https://symfony.com/doc/current/messenger.html#transport-configuration
    async:
      dsn: '%env(MESSENGER_TRANSPORT_DSN)%'
      options:
        use_notify: true
        check_delayed_interval: 60000
      retry_strategy:
        max_retries: 3
        multiplier: 2
    failed: 'doctrine://default?queue_name=failed'
    # sync: 'sync://'
```

Si un problème survient lors de la manipulation d'un message, le consumer réessaiera 3 fois avant d'abandonner. Mais au lieu de jeter le message, il le stockera indéfiniment dans la file d'attente `failed`, qui utilise une autre table de la base de données.

- ➔ Inspectez les messages ayant échoué et relancez-les à l'aide des commandes suivantes :

```
symfony console messenger:failed:show  
symfony console messenger:failed:retry
```

- ➔ Commitez votre travail via Git :

```
git add .  
git commit -m "Add asynchronous processing"
```

#### Liens utiles :

- [Messenger Component](#)

# 13. Symfony Workflow

## Prendre des décisions avec un workflow

Avoir un état pour un modèle est assez commun. L'état du commentaire n'est déterminé que par le vérificateur de spam. Et si on ajoutait d'autres critères de décision ?

Nous pourrions laisser l'admin du site modérer tous les commentaires après le vérificateur de spam. Le processus serait quelque chose comme :

- Commencez par un état `submitted` lorsqu'un commentaire est soumis par un internaute ;
- Laissez le vérificateur de spam analyser le commentaire et changer l'état en `potential_spam`, `ham` ou `rejected`
- S'il n'est pas rejeté, attendez que l'admin du site décide si le commentaire est suffisamment utile en changeant l'état pour `published` ou `rejected`.

La mise en œuvre de cette logique n'est pas trop complexe, mais vous pouvez imaginer que l'ajout de règles supplémentaires augmenterait considérablement la complexité. Au lieu de coder la logique nous-mêmes, nous pouvons utiliser le composant Symfony Workflow :

- ➔ **Installez le composant Workflow :**

```
symfony composer req workflow
```

# Définir un workflow

- ➡ Le workflow de commentaires peut être décrit dans le fichier `config/packages/workflow.yaml` :

```
framework:
  workflows:
    comment:
      type: state_machine
      audit_trail:
        enabled: "%kernel.debug%"
      marking_store:
        type: 'method'
        property: 'stateAsString'
      supports: [App\Entity\Comment]
      initial_marking: submitted
      places: [submitted, ham, potential_spam, spam, rejected, public]
      transitions:
        accept:
          from: submitted
          to: ham
        might_be_spam:
          from: submitted
          to: potential_spam
        reject_spam:
          from: submitted
          to: spam
        publish:
          from: potential_spam
          to: published
        reject:
          from: potential_spam
          to: rejected
        publish_ham:
          from: ham
          to: published
        reject_ham:
          from: ham
          to: rejected
```

- ➡ Ajouter l'accesseur et le mutateur sur notre champ `state` en tant que méthode `getStateAsString()` et `setStateAsString()` dans la classe `Comment` :

```
public function getStateAsString(): string
{
    return $this->state->value;
}

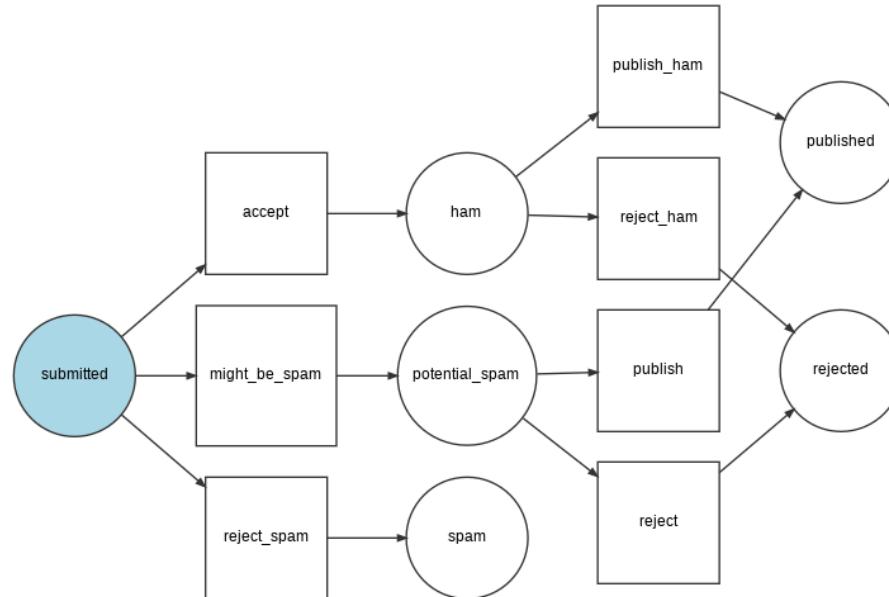
public function setStateAsString(string $state): void
{
    $this->state = CommentStateEnum::from($state);
}
```

- ➡ Mettons à jour notre class enum `CommentStateEnum` avec les nouveaux `state` possible :

```
enum CommentStateEnum : string
{
    case Submitted = 'submitted';
    case Ham = 'ham';
    case PotentialSpam = 'potential_spam';
    case Spam = 'spam';
    case Rejected = 'rejected';
    case Published = 'published';
}
```

- ▶ Pour valider le workflow, générez une représentation visuelle :

```
symfony console workflow:dump comment | dot -Tpng -o workflow.png
```



La commande `dot` fait partie de l'utilitaire [Graphviz](#).

# Utiliser le workflow

- ▶ Remplacez la logique actuelle dans le gestionnaire de messages [src/MessageHandler/CommentMessageHandler.php](#) par le workflow :

```
use Doctrine\ORM\EntityManagerInterface;
+use Psr\Log\LoggerInterface;
use Symfony\Component\Messenger\Attribute\AsMessageHandler;
+use Symfony\Component\Messenger\MessageBusInterface;
+use Symfony\Component\Workflow\WorkflowInterface;

#[AsMessageHandler]
final readonly class CommentMessageHandler
@@ -15,6 +18,9 @@ class CommentMessageHandler
    private EntityManagerInterface $entityManager,
    private SpamChecker $spamChecker,
    private CommentRepository $commentRepository,
+   private MessageBusInterface $bus,
+   private WorkflowInterface $commentStateMachine,
+   private ?LoggerInterface $logger = null,
) { }
```

```
@@ -25,12 +31,18 @@ class CommentMessageHandler
    return;
}

-if (2 === $this->spamChecker->getSpamScore($comment, $message->
-    $comment->setState('spam'));
-} else {
-    $comment->setState('published');
+if ($this->commentStateMachine->can($comment, 'accept')) {
+    $score = $this->spamChecker->getSpamScore($comment, $message->
+        $transition = match ($score) {
+            2 => 'reject_spam',
+            1 => 'might_be_spam',
+            default => 'accept',
+        };
+        $this->commentStateMachine->apply($comment, $transition);
+        $this->entityManager->flush();
+        $this->bus->dispatch($message);
+    } elseif ($this->logger) {
+        $this->logger->debug('Dropping comment message', ['comment' =>
+    }

-
```

La nouvelle logique se lit comme ceci :

- Si la transition `accept` est disponible pour le commentaire dans le message, vérifiez si c'est un spam ;

```
if ($this->commentStateMachine->can($comment, 'accept')) {
```

- Selon le résultat, choisissez la bonne transition à appliquer ;

```
$transition = match ($score) {
    2 => 'reject_spam',
    1 => 'might_be_spam',
    default => 'accept',
};
```

- Appelez `apply()` pour mettre à jour le `Comment` via un appel à la méthode `setStateAsString()` ;

```
$this->commentStateMachine->apply($comment, $transition);
```

- Appelez `flush()` pour valider les changements dans la base de données ;

```
$this->entityManager->flush();
```

- Réexpédiez le message pour permettre au workflow d'effectuer une nouvelle transition.

```
$this->bus->dispatch($message);
```

Comme nous n'avons pas implémenté la fonctionnalité de validation par l'admin, la prochaine fois que le message sera consommé, le message "Dropping comment message" sera enregistré.

- ➔ Mettons en place une validation automatique en attendant le prochain chapitre dans `src/MessageHandler/CommentMessageHandler.php`:

```
$this->commentStateMachine->apply($comment, $transition);
$this->entityManager->flush();
$this->bus->dispatch($message);

+
+    } elseif ($this->commentStateMachine->can($comment, 'publish')
+        || $this->commentStateMachine->can($comment, 'publish_ham')) {
+
+        $this->commentStateMachine->apply($comment, $this->commentStateMachine->can($comment, 'publish') ? 'publish' : 'publish_ham');
+        $this->entityManager->flush();
+
    } elseif ($this->logger) {
        $this->logger->debug('Dropping comment message', ['comment' => $comment->getId(), 'state' => $comment->getState()]);
    }
}
```

- ➔ Exécutez symfony `server:log` et ajoutez un commentaire sur le site pour voir toutes les transitions se produire les unes après les autres.
- ➔ Commitez votre travail via Git :

```
git add .
git commit -m "Add workflow"
```

# Trouver des services depuis le conteneur d'injection de dépendances

Quand nous utilisons l'injection de dépendances, nous récupérons des services depuis le conteneur d'injection de dépendances en utilisant le typage par interface ou parfois par une implémentation de classe concrète. Mais quand une interface à plusieurs implementations, Symfony ne peut deviner celle dont vous avez besoin. Nous avons besoin d'être explicite.

Nous venons juste de rencontrer un cas semblable avec l'injection de `WorkflowInterface` dans la section précédente.

Comme nous injectons n'importe quelle instance de l'interface générique `WorkflowInterface` dans le constructeur, comment Symfony peut savoir quelle implementation du workflow utiliser ? Symfony utilise une convention basée sur le nom de l'argument : `$commentStateMachine` fait référence au workflow comment de la configuration (dont le type est `state_machine`). Essayez n'importe quel autre argument et l'injection échouera.

- ▶ Si vous ne vous rappelez pas de la convention, utilisez la commande `debug:container`. Cherchez tous les services contenant "`workflow`" :

```
symfony console debug:container workflow
Select one of the following services to display its information:
...
[8] Symfony\Component\Workflow\WorkflowInterface $commentStateMachine
[9] Psr\Log\LoggerInterface $workflowLogger
>
```

Remarquez le choix 8, `Symfony\Component\Workflow\WorkflowInterface $commentStateMachine` qui vous indique qu'utiliser `$commentStateMachine` comme argument nommé a une signification particulière.

# 14. Emails

## Envoyer des emails aux admins

Pour garantir des commentaires de haute qualité, l'administrateur doit modérer tous les commentaires. Lorsqu'un commentaire est à l'état `ham` ou `potential_spam`, un e-mail doit être envoyé à l'administrateur avec deux liens : un pour accepter le commentaire et un pour le rejeter.

Pour stocker l'email de l'admin, utilisez un paramètre de conteneur. Pour l'exemple, nous autorisons également son paramétrage grâce à une variable d'environnement (ce qui ne devrait pas être nécessaire dans la "vraie vie") :

- ➔ Ajoutez le paramètre `admin_email` dans le fichier `config/services.yaml` :

```
parameters:  
  photo_dir: "%kernel.project_dir%/public/uploads/photos"  
  +  default_admin_email: admin@example.com  
  +  admin_email: "%env(string:default:default_admin_email:ADMIN_EMAIL)%"
```

Une variable d'environnement peut-être "traitée" avant d'être utilisée. Ici, nous utilisons le processeur `default` afin d'utiliser la valeur du paramètre `default_admin_email` si la variable d'environnement `ADMIN_EMAIL` n'existe pas.

# Envoyer une notification par email

Pour envoyer un email, vous pouvez choisir entre plusieurs classes abstraites de `Email`; de `Message`, le niveau le plus bas, à `NotificationEmail`, le niveau le plus élevé. C'est probablement la classe `Email` que vous utiliserez le plus, mais `NotificationEmail` est le choix idéal pour les e-mails internes.

- ➔ Ajoutez l'envoi d'un email dans le gestionnaire de messages `src/MessageHandler/CommentMessageHandler.php` :

```
    private WorkflowInterface $commentStateMachine,
+    private MailerInterface $mailer,
+    #[Autowire('%admin_email%')] private string $adminEmail,
    private ?LoggerInterface $logger = null,
) {
@{ ...
} elseif ($this->commentStateMachine->can($comment, 'publish') || $this->commentStateMachine->can($comment, 'publish_ham')) {
-    $this->commentStateMachine->apply($comment, $this->commentStateMachine->can($comment, 'publish') ? 'publish' : 'publish_ham');
-    $this->entityManager->flush();
+    $this->mailer->send((new NotificationEmail()
+        ->subject('New comment posted')
+        ->htmlTemplate('emails/comment_notification.html.twig')
+        ->from($this->adminEmail)
+        ->to($this->adminEmail)
+        ->context(['comment' => $comment])
+    );
}
```

Le `MailerInterface` est notre service de messagerie. Il est capable d'envoyer des emails. Nous utilisons la méthode `send()` pour envoyer un email.

Pour envoyer un email, nous avons besoin d'un expéditeur (l'en-tête `From` / `Sender`). Au lieu de le définir explicitement sur l'instance Email, définissez-le globalement :

- ➔ Définissez l'expéditeur par défaut dans le fichier `config/packages/mail.yaml` :

```
framework:  
  mailer:  
    dsn: '%env(MAILER_DSN)%'  
    envelope:  
      sender: "%admin_email%"
```

# Personnaliser le contenu de l'email

Pour personnaliser le contenu de l'email, créez un modèle Twig dans le répertoire `templates/emails/comment_notification.html.twig` :

- ➔ Créez le modèle Twig `templates/emails/comment_notification.html.twig` :

```
{% extends '@email/default/notification/body.html.twig' %}

{% block content %}
Author: {{ comment.author }}<br />
Email: {{ comment.email }}<br />
State: {{ comment.state }}<br />

<p>
{{ comment.text }}
</p>
{% endblock %}

{% block action %}
<spacer size="16"></spacer>
<button href="{{ url('review_comment', { id: comment.id }) }}">Accept</button>
<button href="{{ url('review_comment', { id: comment.id, reject: true }) }}">Reject</button>
{% endblock %}
```

Le template remplace quelques blocs pour personnaliser le message de l'e-mail et ajouter des liens permettant à l'administrateur d'accepter ou de rejeter un commentaire. Tout argument de route qui n'est pas un paramètre de route valide est ajouté en tant qu'élément de chaîne de requête (l'URL de rejet ressemble à `/admin/comment/review/42?reject=true`).

Le `NotificationEmail` utilise par défaut `Inky` au lieu de `HTML` pour concevoir les e-mails. Il permet de créer des e-mails réactifs compatibles avec tous les clients de messagerie populaires.

Pour une compatibilité maximale avec les lecteurs de courrier électronique, la mise en page de base de notification intègre toutes les feuilles de style (via le package `CSS inliner`) par défaut.

Ces deux fonctionnalités font partie des extensions Twig facultatives qui doivent être installées :

- ➔ Installez les extensions Twig pour Inky et CSS inliner :

```
symfony composer req "twig/cssinliner-extra:^3" "twig/inky-extra:^3"
```

## Générer des URL absolues dans une commande Symfony

Dans les e-mails, on génère des URL avec `url()` au lieu de `path()` pour obtenir des URL absolues (avec schéma et hôte).

L'e-mail est envoyé depuis le gestionnaire de messages, dans un contexte de console. Générer des URL absolues dans un contexte Web est plus facile, car nous connaissons le schéma et le domaine de la page actuelle. Ce n'est pas le cas dans un contexte de console.

- ➔ Définissez le nom de domaine et le schéma à utiliser explicitement :

```
+ admin_email: "%env(string:default:default_admin_email:ADMIN_EMAIL)%"
+   default_base_url: 'http://127.0.0.1'
+   router.request_context.base_url: '%env(default:default_base_url:SYMFONY_DEFAULT_ROUTE_URL)%'
```

# Ajouter la route de review\_comment

- ▶ Ajoutez le controller `src/Controller/Admin/CommentReviewController.php` via le maker:

```
symfony console make:controller 'Admin\ReviewComment' --invokable --no-interaction
```

- ▶ Modifiez le controller pour accepter ou rejeter un commentaire :

```
#[Route('/admin/review/comment/{id}', name: 'review_comment')]
public function __invoke(
    Request $request,
    Comment $comment,
    WorkflowInterface $commentStateMachine,
    EntityManagerInterface $entityManager,
    MessageBusInterface $bus,
): Response {
    $accepted = !$request->query->getBoolean('reject');

    if ($commentStateMachine->can($comment, 'publish')) {
        $transition = $accepted ? 'publish' : 'reject';
    } elseif ($commentStateMachine->can($comment, 'publish_ham')) {
        $transition = $accepted ? 'publish_ham' : 'reject_ham';
    } else {
        return new Response('Comment already reviewed or not in the right state.');
    }

    $commentStateMachine->apply($comment, $transition);
    $entityManager->flush();

    if ($accepted) {
        $bus->dispatch(new CommentMessage($comment->getId()));
    }

    return $this->render('admin/review_comment.html.twig', ['transition' => $transition, 'comment' => $comment, ]);
}
```

L'URL du commentaire de révision commence par `/admin/` pour la protéger avec le pare-feu défini lors d'une étape précédente. L'administrateur doit être authentifié pour accéder à cette ressource.

Une fois l'examen terminé, un court modèle remercie l'administrateur pour son travail acharné :

- ➔ Créez le modèle Twig `templates/admin/review.html.twig` :

```
{% extends 'base.html.twig' %}

{% block title %}Revue de commentaire{% endblock %}

{% block body %}
    <h2>Comment reviewed, thank you!</h2>

    <p>Applied transition: <strong>{{ transition }}</strong></p>
    <p>New state: <strong>{{ comment.state }}</strong></p>
{% endblock %}
```

## Utiliser un mail catcher

Au lieu d'utiliser un "vrai" serveur SMTP ou un fournisseur tiers pour envoyer des e-mails, utilisons un **mail catcher**.

 Un mail catcher fournit un serveur SMTP qui ne délivre pas les e-mails, mais les rend disponibles via une interface Web.

Heureusement, Symfony a déjà configuré automatiquement un tel outil pour nous :

```
# compose.override.yaml

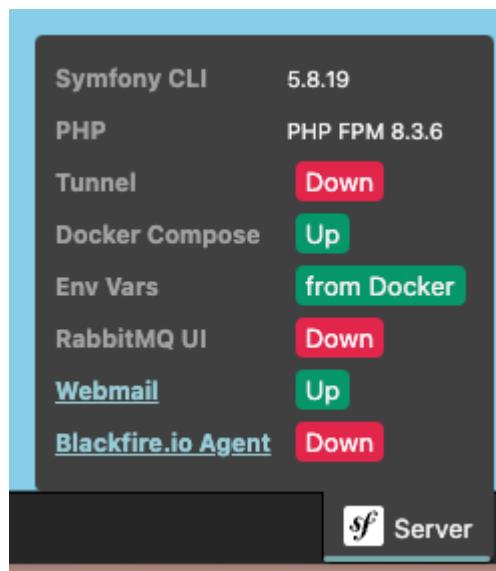
mailer:
    image: axllent/mailpit
    ports:
        - "1025"
        - "8025"
    environment:
        MP_SMTP_AUTH_ACCEPT_ANY: 1
        MP_SMTP_AUTH_ALLOW_INSECURE: 1
```

# Accéder au Webmail

- ▶ Démarrez le mail catcher :

```
symfony open:local:webmail
```

Ou depuis la barre d'outils de débogage Web :



Depuis gitpod, vous pouvez accéder à l'interface Webmail via le lien fourni dans la console.

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL	PORTS	COMMENTS	⋮	X				
						Port	Address	State			
●	8000					<a href="https://8000-faharidev-s...">https://8000-faharidev-s...</a>					open (private)
●	5432					<a href="https://5432-faharidev-s...">https://5432-faharidev-s...</a>					open (private)
●	32768					<a href="https://32768-faharidev-...">https://32768-faharidev-...</a>					open (private)
●	32769					<a href="https://32769-faharidev-...">https://32769-faharidev-...</a>					open (private)
●	36873					<a href="https://36873-faharidev-...">https://36873-faharidev-...</a>					open (private)
●	42395					<a href="https://42395-faharidev-...">https://42395-faharidev-...</a>					open (private)

# Test des e-mails

Il existe de nombreuses façons de tester les e-mails. Vous pouvez écrire des tests unitaires si vous écrivez une classe par email (qui étend de `Email` ou `TemplatedEmail` par exemple).

Les tests les plus courants que vous écrirez sont cependant des tests fonctionnels qui vérifient que certaines actions déclenchent un e-mail, et probablement des tests sur le contenu des e-mails s'ils sont dynamiques.

Symfony est livré avec des assertions qui facilitent de tels tests, voici un exemple de test qui démontre quelques possibilités :

```
public function testMailerAssertions()
{
    $client = static::createClient();
    $client->request('GET', '/');

    $this->assertEmailCount(1);
    $event = $this->getMailerEvent(0);
    $this->assertEmailIsQueued($event);

    $email = $this->getMailerMessage(0);
    $this->assertEmailHeaderSame($email, 'To', 'fabien@example.com');
    $this->assertEmailTextBodyContains($email, 'Bar');
    $this->assertEmailAttachmentCount($email, 1);
}
```

Ces assertions fonctionnent lorsque les e-mails sont envoyés de manière synchrone ou asynchrone.

- ➔ Commitez votre travail via Git :

```
git add .
git commit -m "Add email notifications"
```

# 15. Mise en cache pour les performances

## Ajout d'en-têtes de cache HTTP

Des problèmes de performances peuvent survenir avec la popularité. Quelques exemples typiques : index de base de données manquants ou tonnes de requêtes SQL par page. Vous n'aurez aucun problème avec une base de données vide, mais avec plus de trafic et de données croissantes, cela pourrait survenir à un moment donné.

L'utilisation de stratégies de mise en cache `HTTP` est un excellent moyen d'optimiser les performances pour les utilisateurs finaux avec peu d'effort. Ajoutez un cache proxy inverse en production pour activer la mise en cache et utilisez un [https://en.wikipedia.org/wiki/Content\\_delivery\\_network](https://en.wikipedia.org/wiki/Content_delivery_network) pour mettre en cache en périphérie pour des performances encore meilleures.

- ➔ Mettons en cache la page d'accueil pendant une heure :

```
return $this->render('conference/index.html.twig', [
    'conferences' => $conferenceRepository->findAll(),
    -]);
+   ])->setSharedMaxAge(3600);
}]
```

La méthode `setSharedMaxAge()` configure l'expiration du cache pour les **reverse proxy**. Utilisez `setMaxAge()` pour contrôler le cache du navigateur. Le temps est exprimé en secondes (1 heure = 60 minutes = 3600 secondes).

La mise en cache de la page de la conférence est plus difficile, car plus dynamique. N'importe qui peut ajouter un commentaire à tout moment et personne ne veut attendre une heure pour le voir en ligne. Dans de tels cas, utilisez la stratégie de validation `HTTP`.

# Activation du Symfony HTTP Cache Kernel

Pour tester la mise en cache, vous pouvez activer le [Symfony HTTP Cache Kernel](#) mais seulement en développement (pour l'environnement "production", nous utiliserons une solution "plus robuste").

- ▶ Ajouter le cache en dev dans le fichier `config/packages/framework.yaml` :

```
session:  
    storage_factory_id: session.storage.factory.mock_file  
+  
+when@dev:  
+    framework:  
+        http_cache: true
```

En plus d'être un HTTP reverse proxy à part entière, le HTTP reverse proxy de Symfony (via la classe `HttpCache`) ajoute de belles informations de débogage sous forme d'en-têtes HTTP. Cela aide grandement à valider les en-têtes de cache que nous avons définis.

- ▶ Vérifiez-le sur la page d'accueil :

```
curl -s -I -X GET https://127.0.0.1:8000/  
  
HTTP/2 200  
age: 0  
cache-control: public, s-maxage=3600  
content-type: text/html; charset=UTF-8  
date: Mon, 28 Oct 2019 08:11:57 GMT  
x-content-digest: en63cef7045fe418859d73668c2703fb1324fcc0d35b21d95369a9ed1aca48e73e  
x-debug-token: 9eb25a  
x-debug-token-link: https://127.0.0.1:8000/_profiler/9eb25a  
x-robots-tag: noindex  
x-symfony-cache: GET /: miss, store  
content-length: 50978
```

# 16. Styliser l'interface utilisateur avec Webpack

## Installer Webpack Encore

Nous n'avons pas consacré de temps à la conception de l'interface utilisateur. Pour styliser comme un pro, nous utiliserons une stack moderne, basée sur Webpack. Et pour ajouter une touche Symfony et faciliter son intégration avec l'application, utilisons Webpack Encore :

- ➔ **Installez Webpack Encore et supprimer Asset Mapper :**

```
symfony composer rem asset-mapper  
symfony composer req encore
```

Un environnement Webpack complet a été créé pour vous : package.json et webpack.config.jsa été généré et contient une bonne configuration par défaut. Open webpack.config.js, il utilise l'abstraction Encore pour configurer Webpack.

Le fichier `package.json` définit quelques commandes intéressantes que nous utiliserons tout le temps.

Le répertoire `assets` contient les principaux points d'entrée des actifs du projet : `styles/app.css` et `app.js`.

# Utiliser Sass

- ▶ Au lieu d'utiliser du CSS simple, passons à **Sass** :

```
mv assets/styles/app.css assets/styles/app.scss
```

*assets/app.js*

```
# assets/app.js  
  
// any CSS you import will output into a single css file (app.css in this case)  
-import './styles/app.css';  
+import './styles/app.scss';
```

- ▶ Installez le chargeur **Sass** :

```
npm install node-sass sass-loader@13 --save-dev
```

- ▶ Et activez le chargeur **Sass** dans **webpack.config.js**

```
// enables Sass/SCSS support  
-  //enableSassLoader()  
+  .enableSassLoader()
```

Comment savoir quels packages installer ? Si nous avions essayé de créer nos actifs sans eux, Encore nous aurait donné un joli message d'erreur suggérant la commande `npm install` nécessaire pour installer les dépendances pour charger les fichiers `.scss`.

# Tirer parti de Bootstrap

Pour commencer avec de bonnes valeurs par défaut et créer un site Web réactif, un framework CSS comme Bootstrap peut aller très loin. Installez-le sous forme de package :

- ➔ **Installez Bootstrap :**

```
npm install bootstrap @popperjs/core bs-custom-file-input --save-dev
```

- ➔ **Exiger Bootstrap dans le fichier CSS `assets/styles/app.scss` (nous avons également nettoyé le fichier) :**

```
-body {  
-  background-color: lightgray;  
-}  
+@import '~bootstrap/scss/bootstrap';
```

- ➔ **Faites de même pour le fichier JS `assets/app.js` :**

```
// any CSS you import will output into a single css file (app.css in this case)  
import './styles/app.scss';  
+import 'bootstrap';  
+import bsCustomFileInput from 'bs-custom-file-input';  
+  
+bsCustomFileInput.init();
```

- ➔ **Activer le thème Bootstrap dans le fichier `config/packages/twig.yaml` :**

```
twig:  
  form_themes: ['bootstrap_5_layout.html.twig']
```

## Styliser le HTML

Nous sommes maintenant prêts à styliser l'application.

-  Téléchargez et développez l'archive à la racine du projet :

```
php -r "copy('https://symfony.com/uploads/assets/guestbook-6.4.zip', 'guestbook-6.4.zip');"  
unzip -o guestbook-6.4.zip  
rm guestbook-6.4.zip
```

Jetez un œil aux modèles, vous apprendrez peut-être une astuce ou deux sur Twig.

## Construisons Assets

Un changement majeur lors de l'utilisation de Webpack est que les fichiers CSS et JS ne sont pas utilisables directement par l'application. Ils doivent d'abord être "compilés".

- ▶ En développement, la compilation des actifs peut se faire via la commande `encore dev` :

```
symfony run npm run dev
```

Au lieu d'exécuter la commande à chaque fois qu'il y a un changement,

- ▶ envoyez-la en arrière-plan et laissez-la surveiller les modifications JS et CSS :

```
symfony run -d npm run watch
```

Prenez le temps de découvrir les changements visuels. Jetez un œil au nouveau design dans un navigateur.