



## Symfony #2

▼ Version	V1
≡ Type	Technique
📅 Date de création	@14 décembre 2023 15:35
📅 Dernière modification	@14 décembre 2023 15:40

### ? Les formulaires

[🔗 Documentation Symfony formulaires](#)

**Le flux de travail recommandé lorsque l'on travail avec des formulaires Symfony est le suivant :**

1. construire le formulaire en utilisant une classe de formulaire
2. Afficher le formulaire dans un template afin que l'utilisateur puisse interagir avec
3. Traiter le formulaire pour valider les données soumises, transformez-le en données et faites quelque chose avec ( par exemple, persistez-le dans une base de données ).

#### ▼ Créer un formulaire

Nous utiliserons la commande mise à disposition par le Maker bundle :

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

Le prompt nous demande l'entité pour laquelle nous souhaitons créer un formulaire. Une fois validé un formulaire est généré, ici j'ai généré un `src/Form/DemoType`.

```
<?php

namespace App\Form;

use App\Entity\Demo;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class DemoType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('title')
            ->add('category')
        ;
    }

    public function configureOptions(OptionsResolver $resolver): void
    {
        $resolver->setDefaults([
            'data_class' => Demo::class,
        ]);
    }
}
```

▼ Afficher le formulaire

Dans cet exemple nous utiliserons le contrôleur `src/Controller/DemoController` pour afficher le formulaire sur la route `/demo`.

Pour cela nous utiliserons la méthode `createForm` qui nous a été mise à disposition par la classe `AbstractController` que notre contrôleur étend. La méthode `createForm` prend comme premier argument la classe du formulaire que l'on souhaite utiliser, ici `DemoType` issu du fichier `App\Form/DemoType` qui a été généré précédemment par la commande. Le second argument est une instance de l'entité que l'on va créer grâce au formulaire, ici une instance de la classe `Demo`.

Pour finir on passe la variable contenant notre formulaire au template pour que celui-ci se charge de l'afficher.

```
<?php

namespace App\Controller;

use App\Entity\Demo;
use App\Form\DemoType;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class DemoController extends AbstractController
{
    #[Route('/demo', name: 'app_demo')]
    public function index(): Response
    {
        $demo = new Demo();

        $form = $this->createForm(DemoType::class, $demo);

        return $this->render('demo.html.twig', [
            'form' => $form
        ]);
    }
}
```

```
}  
}
```

Comme indiqué dans la méthode

`render` de mon contrôleur, celui-ci exploite le template `demo.html.twig`, c'est dans ce template que l'on aura accès à la variable `form` contenant notre formulaire.

Pour afficher un formulaire nous pouvons utiliser la méthode Twig `form()` en lui donnant comme argument la variable contenant notre formulaire. Cette méthode se chargera d'afficher le formulaire sans possibilité de gérer l'organisation des balise HTML le composant.

```
{% extends 'base.html.twig' %}  
  
{% block pageTitle %}  
    {{ page_title }}  
{% endblock %}  
  
{{ form(form) }}
```

Pour avoir la main sur l'architecture HTML de notre formulaire nous disposons des méthodes Twig suivantes :

<code>form_start(form)</code>	obligatoire - Indique le début d'un formulaire, prend le formulaire comme argument
<code>form_errors(form)</code>	Affichera les erreurs du formulaire ( pas celles des champs du formulaire ), prend le formulaire comme argument
<code>form_row(form.task)</code>	Affichera un champ du formulaire ( avec son label, champ, erreur, etc ), prend un champ du formulaire comme argument
<code>form_end(form)</code>	obligatoire - Indique la fin d'un formulaire, prend le formulaire comme argument

**Le résultat en Twig :**

Forme minimal pour inclure le bouton de soumission du formulaire

```
{{ form_start(form) }}  
{{ form_row(form.title) }}
```

Forme avancée

```
{{ form_start(form) }}  
    <div class="my-custom-class-for-errors">
```

```

{{ form_row(form.description) }}
{{ form_row(form.city) }}
{{ form_row(form.price) }}
{{ form_row(form.family) }}
{{ form_row(form.category) }}
<button type="submit">Submit</button>
{{ form_end(form) }}

```

```

        {{ form_errors(form) }}
    </div>

    <div class="row">
        <div class="col">
            {{ form_row(form.title) }}
        </div>
        <div class="col" id="some-custom-id">
            {{ form_row(form.description) }}
        </div>
    </div>

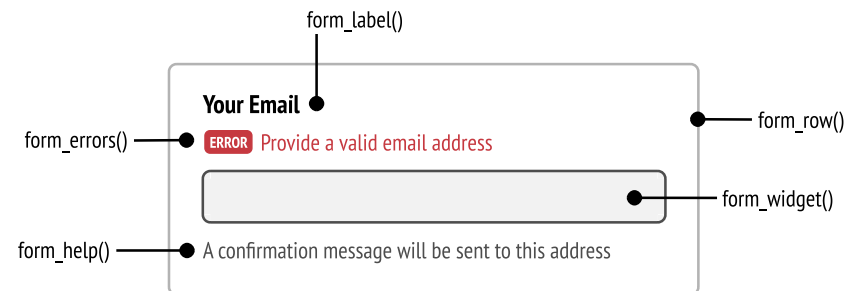
    <button type="submit">Submit</button>
    {{ form_end(form) }}

```

Il est possible d'aller encore plus loin dans l'affichage de son formulaire, la fonction

`form_row()` affiche un champ du formulaire avec son label, ses erreurs, son input, etc .. pour avoir la main sur chaque composant d'un champ de formulaire nous disposons des méthodes Twig suivantes :

<code>form_label(form.dueDate)</code>	Affichera le label d'un champ, prend un champ du formulaire comme argument
<code>form_widget(form.dueDate)</code>	Affichera le widget ( input ) d'un champ, prend un champ du formulaire comme argument
<code>form_help(form.dueDate)</code>	Affichera le texte d'aide d'un champ, prend un champ du formulaire comme argument
<code>form_errors(form.dueDate)</code>	Affichera les erreurs d'un champ, prend un champ du formulaire comme argument



Le résultat en Twig :

```
<div class="form-control">

    <p>{{ form_label(form.title) }}</p>

    {{ form_widget(form.title) }}

    <p>{{ form_help(form.title) }}</p>

    <div class="form-error">
        {{ form_errors(form.title) }}
    </div>

</div>
```

Exemple de formulaire "final" :

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

{% block title %}Création d'une Dynamo{% endblock %}

{% block body %}

    {# V.0 formulaire 🐛 #}
    {# {{ form(form) }} #}

    {# V.1 formulaire 👍 #}
    {# DEBUT DU FORMULAIRE #}
    {#     {{ form_start(form) }} #}
```

```

{# ERREURS DU FORMULAIRE #}
{#     {{ form_errors(form) }} #}
{# CHAMPS DU FORMULAIRE #}
{#     {{ form_row(form.seats) }} #}
{#     {{ form_row(form.description) }} #}
{# BOUTON DE SOUMISSION DU FORMULAIRE #}
{#     <input type="submit" value="Enregistrer"> #}
{# FIN DU FORMULAIRE #}
{#     {{ form_end(form) }} #}

```

```

{# V.2 formulaire 🔥 #}
{# DEBUT DU FORMULAIRE #}
{{ form_start(form) }}

```

```

{# ERREURS DU FORMULAIRE #}
{{ form_errors(form) }}

```

```

{# CHAMPS DU FORMULAIRE #}
<div class="inputWrapper">
    {{ form_label(form.seats) }}
    {{ form_widget(form.seats) }}
    {{ form_errors(form.seats) }}
    {{ form_help(form.seats) }}
</div>

```

```

<div class="inputWrapper">
    {{ form_label(form.description) }}
    {{ form_widget(form.description) }}
    {{ form_errors(form.description) }}

```

```

        {{ form_help(form.description) }}
    </div>

    {# BOUTON DE SOUMISSION DU FORMULAIRE #}
    <input type="submit" value="Enregistrer">

    {# FIN DU FORMULAIRE #}
    {{ form_end(form) }}

{% endblock %}

```

### ▼ Traiter la soumission du formulaire

Nous exploitons le `DemoController` pour afficher le formulaire, c'est au même endroit que nous allons gérer la soumission de celui-ci.

```

<?php

namespace App\Controller;

use App\Entity\Demo;
use App\Form\DemoType;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class DemoController extends AbstractController
{
    #[Route('/demo', name: 'app_demo')]
    public function index(EntityManagerInterface $entityManager, Request $request): Response
    {
        $demo = new Demo();
    }
}

```



```

$form = $this->createForm(DemoType::class, $demo);

    // Ecoute la soumission du formulaire
    $form->handleRequest($request);

    // Condition valide lorsque le formulaire est soumis et valide
    if ($form->isSubmitted() && $form->isValid()) {

        // Persiste les données du formulaire dans l'entité Demo
        $demo = $form->getData();

        // Exécuter la logique que vous souhaitez
        // par exemple enregistrer la nouvelle entité en base de données

        $entityManager->persist($demo);
        $entityManager->flush();
    }

    return $this->render('demo.html.twig', [
        'form' => $form
    ]);
}

```

#### Voici en détail comment fonctionne le formulaire :

1. Lors du chargement initial de la page dans un navigateur, le formulaire n'a pas encore été soumis et `$form->isSubmitted()` renvoie false. Ainsi, le formulaire est créé et rendu visuellement.
2. Lorsque l'utilisateur soumet le formulaire, `handleRequest()` le reconnaît et réécrit immédiatement les données soumises dans les propriétés de l'objet `$demo`. Cet objet est ensuite validé, s'il n'est pas valide `isValid()` renvoie false et le formulaire est rendu à nouveau, mais maintenant avec des erreurs de

validation.

3. Lorsque l'utilisateur soumet le formulaire avec des données valides `isValid()` renvoie true. Nous avons maintenant la possibilité d'effectuer certaines actions à l'aide de l'objet `$demo` ( par exemple, le conserver dans la base de données ) avant de par exemple rediriger l'utilisateur vers une autre page.

#### ▼ Validation des données

 [La liste de toutes les contraintes mises à dispositions par Symfony.](#)

Par mesure de sécurité il est indispensable de contrôler les données soumises par l'utilisateur avant de les enregistrer en base de données, c'est notamment ce que chercher à vérifier la méthode `isValid()` lorsque l'on soumet le formulaire ( voir chapitre précédent ).

Le but de la validation est d'indiquer si les données d'un objet sont valides. Pour que cela fonctionne, nous allons configurer une liste de règles ( appelées contraintes ) que l'objet doit suivre pour être valide.

Ici nous indiquons que le champ `$name` ne peut pas être `null`. L'ajout de cette configuration ne garantit pas encore que la valeur ne sera pas `null`. Pour réellement garantir que la valeur respecte la contrainte, l'objet doit être passé au service de validation pour être vérifié ( `isValid()` lors de la soumission du formulaire ).

```
namespace App\Entity;

// ...
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    #[Assert\NotBlank]
    private $name;
}
```

## L'authentification

 [Documentation authentification Symfony](#)

### ▼ Configuration

L'authentification d'un utilisateur est **grandement facilité par le Security bundle de Symfony**, celui-ci met à disposition tout le nécessaire pour créer notre système d'authentification.

Assurez-vous d'avoir le Security bundle installé dans votre projet ( voir le fichier `symfony.lock` , il contient la liste des bundles Symfony installés sur votre projet ). Autrement installer le :

```
composer require symfony/security-bundle
```

Une fois installé un fichier de configuration est généré `conf/packages/security.yaml` . Ce fichier contient la configuration de votre bundle de sécurité, nous y reviendrons plus tard.

Les autorisations dans Symfony sont toujours liées à un objet utilisateur. Si vous avez besoin de sécuriser ( des parties de ) votre application, vous devez créer une classe d'utilisateurs. Il s'agit d'une classe qui implémente la classe `UserInterface` .

Pour créer cette entité et qu'elle soit automatiquement reconnu par le `SecurityBundle` nous utiliserons la commande mise à disposition par le Maker bundle :

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

### Le prompt :

1. Le nom de l'entité, "User" nous ai proposé → utiliser "User"
2. Nos utilisateurs doivent-ils être stockés dans la base de données ? → oui
3. Le nom de la propriété que l'on utilisé pour identifier notre utilisateur ( exemple : email, username, uuid ... ), "email" nous ai proposé → utiliser "email"

#### 4. Les mots de passes doivent-ils être encryptés ? → oui

À la suite de cette commande une entité `src/Entity/User` a été créée avec les propriétés nécessaires ( id, email, roles ... ) ainsi que le répertoire associé à l'entité `src/Repository/UserRepository` . Vous êtes libres d'ajouter des champs à votre entité `User` .

Le fichier de configuration `conf/packages/security.yaml` à lui aussi été modifié automatiquement, la partie “provider” qui ressemblait à cela :

```
security:
    ...
    providers:
        users_in_memory: { memory: null }
```

Ressemble maintenant à ça :

```
security:
    ...
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
```

N'oubliez pas de créer la migration de base de données et de l'exécuter pour appliquer les changements à la base de données :

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

Visiter une page de votre application, dans la toolbar de Symfony vous devriez dorénavant voir que vous êtes un visiteur non authentifié, et que la page sur laquelle vous vous trouvez est protégé par le firewall

`main`, que l'on retrouve dans notre fichier de configuration.



On constate que le firewall `main` utilise le provider `app_user_provider` qui lui même s'appuie sur notre entité `User` pour authentifier un utilisateur. Un firewall peut utiliser plusieurs provider à la fois.

```
security:
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
```

```
lazy: true
provider: app_user_provider
```

### ▼ Formulaire de connexion

Nous utiliserons la commande mise à disposition par le Maker bundle.

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

#### Le prompt :

1. Quel nom souhaitons nous donner à notre “authenticator” → `UserAuthenticator` semble adapté, libre à vous de choisir
2. Le système de connexion à besoin d'un contrôleur, il nous a demandé quel nom souhaitons nous donner à ce contrôleur → la proposition du prompt, `SecurityController` semble adapté, libre à vous de choisir
3. Est-ce que l'on souhaite avoir une route `/logout` destiné à la déconnexion → oui

À la suite de cette commande les fichiers suivants sont modifiés/générés :

généré	<code>src/Security/UserAuthenticator.php</code>	La logique pour l'authentification, ⚠ penser à changer la redirection ! ligne 49, 50, 51
généré	<code>src/Controller/SecurityController.php</code>	Le contrôleur contenant les routes <code>login</code> et <code>logout</code>
généré	<code>templates/security/login.html.twig</code>	Le template avec le formulaire de connexion
modifié	<code>config/packages/security.yaml</code>	Le fichier de configuration de la sécurité est mis à jour pour utiliser notre <code>UserAuthenticator.php</code>

### ▼ Formulaire d'inscription

Nous utiliserons la commande mise à disposition par le Maker bundle.

```
php bin/console make:registration-form
```

#### Le prompt :

1. Est-ce que l'on souhaite que nos utilisateurs soient uniques, cela rajoute une contrainte sur le champ "email" du `User` pour qu'il ne puisse pas y avoir de doublon d'adresse email → oui
2. Est-ce que l'on souhaite envoyer un email de vérification lors de l'inscription de l'utilisateur → dans le cadre du cours non
3. Est-ce que l'on souhaite que l'utilisateur soit authentifié ( connecté ) après son enregistrement → oui
4. Choisir la route sur laquelle on souhaite rediriger l'utilisateur une fois qu'il c'est enregistré

À la suite de cette commande l'entité `User` est modifié si vous avez accepté que le champ d'identifiant de vos utilisateurs soit unique, les fichiers `src/Form/RegistrationForm`, `src/Controller/RegistrationForm` et `templates/registration/register.html.twig` ont étaient générés. Vous êtes libre de modifier le code, le déplacer, etc.

<code>src/Form/RegistrationForm</code>	La classe du formulaire d'inscription
<code>src/Controller/RegistrationForm</code>	La route du formulaire ainsi que son traitement
<code>templates/registration/register.html.twig</code>	L'affichage client du formulaire

#### ▼ Tester l'authentification

 **Pré-requis** : avoir un système d'authentification configuré + le formulaire de connexion

L'objectif est de créer un utilisateur par le biais de PHPMyAdmin ou grâce à une requête SQL et ensuite se connecter à l'aide de ces identifiants pour vérifier que le système d'authentification fonctionne.

### Que vous utilisiez PHPMyAdmin, une requête SQL ou n'importe qu'elle autre technique il va être nécessaire d'encrypter le mot de passe de l'utilisateur que vous créez. Pourquoi ?

Lorsque l'utilisateur s'inscrit "normalement", par le biais du formulaire d'inscription, son mot de passe n'est pas stocké en "clair" dans la base de données. On ne sauvegarde que la version encrypté du mot de passe, par sécurité.

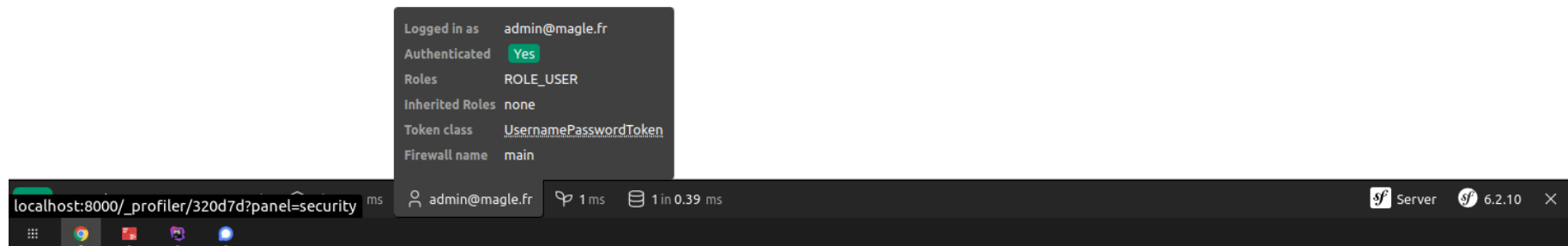
Lorsque l'utilisateur soumet le formulaire de connexion Symfony va dans un premier temps vérifier que l'email correspond bien à un utilisateur, dans un second il vérifiera que les mots de passes correspondent. Pour cela il va encrypter le mot de passe soumis dans le formulaire puis le comparer le mot de passe sauvegardé en base de données qui est déjà encrypté.

En brief, si en base de données votre utilisateur à le mot de passe 123456 , saisir 123456 dans le formulaire ne fonctionnera pas car Symfony va comparé la version du mot de passe issu de la base de données ( **qui devrait être encrypté** ) 123456 , à la version encrypté du mot de passe soumis dans le formulaire. Pour information voici à ce quoi peu ressembler la version encrypté du mot de passe 123456 : \$2y\$13\$jSp1PaBgYxwx0nuFBbUdu0tYJqs.jtFNzDC/CPG3ALzuGB4vK/khS .

Pour encrypter un mot de passe manuellement, vous pouvez utiliser la commande suivante et lui indiquer le mot de passer à encrypter. Vous obtiendrez la version encrypté du mot de passe, ainsi vous pouvez créer un utilisateur en base de données avec l'adresse email que vous souhaitez et le mot de passe fraîchement encrypté. Vous n'aurez plus qu'a utiliser l'email et le mot de passe en clair dans le formulaire de connexion, à l'issue de quoi vous devrez être connecté dans la toolbar Symfony.

```
php bin/console security:hash-password
```





## 👂 Écouter les événements Symfony

[Documentation Symfony eventSubscriber et eventListener](#)

Les `eventSubscriber` et les `eventListener` vont nous permettre d'ajouter des actions au déclenchement d'événement précis. Prenons l'exemple de la connexion / déconnexion de notre utilisateur, toute cette logique est assurée par Symfony, si nous souhaitons ajouter des actions lors de l'un de ces événements nous allons utiliser un `eventSubscriber` ou un `eventListener` qui s'exécutera lorsque l'événement de connexion ou de déconnexion se déclenchera. Cela permet entre autre de personnaliser les comportements de l'application.

Les `eventSubscriber` et les `eventListener` peuvent être utilisés indistinctement dans la même application. La décision d'utiliser l'un ou l'autre est généralement une question de goût. Cependant, il y a quelques avantages mineurs pour chacun d'eux :

- Les `eventSubscriber` sont plus faciles à réutiliser car la connaissance des événements est conservée dans la classe plutôt que dans la définition du service. C'est la raison pour laquelle Symfony utilise les abonnés en interne.
- Les `eventListener` sont plus flexibles car les bundles peuvent activer ou désactiver chacun d'eux de manière conditionnelle en fonction d'une certaine valeur de configuration.

Dans la suite de cette présentation nous créerons un `eventSubscriber`, pour cela nous utiliserons le Maker bundle avec la commande suivante pour générer notre `eventSubscriber` :

```
php bin/console make:subscriber
```

Après avoir donné un nom à notre `eventSubscriber` le prompt affiche une liste d'événements susceptibles de nous intéresser, pour lui indiquer les événements de la liste que nous souhaitons écouter copier la partie blanche de la ligne dans le terminal puis appuyer sur entrée. **Vous ne pouvez n'en sélectionner qu'un.** Dans l'image suivante j'indique que je souhaite écouter l'événement `Symfony\Component\Security\Http\Event\LoginSuccessEvent` qui correspond à l'événement se déclenchant à la connexion de l'utilisateur.

```
~/Simplon/demo_symfony | master ! ?
> php bin/console make:subscriber

Choose a class name for your event subscriber (e.g. ExceptionSubscriber):
> DemoSubscriber

Suggested Events:
* Symfony\Component\Mailer\Event\MessageEvent (Symfony\Component\Mailer\Event\MessageEvent)
* Symfony\Component\Messenger\Event\WorkerMessageFailedEvent (Symfony\Component\Messenger\Event\WorkerMessageFailedEvent)
* Symfony\Component\Messenger\Event\WorkerMessageHandledEvent (Symfony\Component\Messenger\Event\WorkerMessageHandledEvent)
* Symfony\Component\Messenger\Event\WorkerRunningEvent (Symfony\Component\Messenger\Event\WorkerRunningEvent)
* Symfony\Component\Messenger\Event\WorkerStartedEvent (Symfony\Component\Messenger\Event\WorkerStartedEvent)
* Symfony\Component\Notifier\Event\MessageEvent (Symfony\Component\Notifier\Event\MessageEvent)
* Symfony\Component\Security\Http\Event\CheckPassportEvent (Symfony\Component\Security\Http\Event\CheckPassportEvent)
* Symfony\Component\Security\Http\Event>LoginSuccessEvent (Symfony\Component\Security\Http\Event>LoginSuccessEvent)
* Symfony\Component\Security\Http\Event\LogoutEvent (Symfony\Component\Security\Http\Event\LogoutEvent)
* console.command (Symfony\Component\Console\Event\ConsoleCommandEvent)
* console.error (Symfony\Component\Console\Event\ConsoleErrorEvent)
* console.terminate (Symfony\Component\Console\Event\ConsoleTerminateEvent)
* debug.security.authorization.vote (Symfony\Component\Security\Core\Event\VoteEvent)
* kernel.controller (Symfony\Component\HttpKernel\Event\ControllerEvent)
* kernel.controller_arguments (Symfony\Component\HttpKernel\Event\ControllerArgumentsEvent)
* kernel.exception (Symfony\Component\HttpKernel\Event\ExceptionEvent)
* kernel.finish_request (Symfony\Component\HttpKernel\Event\FinishRequestEvent)
* kernel.request (Symfony\Component\HttpKernel\Event\RequestEvent)
* kernel.response (Symfony\Component\HttpKernel\Event\ResponseEvent)
* kernel.terminate (Symfony\Component\HttpKernel\Event\TerminateEvent)
* kernel.view (Symfony\Component\HttpKernel\Event\ViewEvent)
* security.authentication.success (Symfony\Component\Security\Core\Event\AuthenticationSuccessEvent)
* security.interactive_login (Symfony\Component\Security\Http\Event\InteractiveLoginEvent)
* security.switch_user (Symfony\Component\Security\Http\Event\SwitchUserEvent)

What event do you want to subscribe to?:
> Symfony\Component\Security\Http\Event>LoginSuccessEvent
```

Le fichier `src/EventSubscriber/DemoSubscriber` est créé et contient le code suivant. Il est nécessaire que notre subscriber implements la classe `EventSubscriberInterface`.

```

<?php

namespace App\EventSubscriber;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Security\Http\Event\LoginSuccessEvent;

class DemoSubscriber implements EventSubscriberInterface
{
    public function onLoginSuccessEvent(LoginSuccessEvent $event): void
    {
        // ...
    }

    public static function getSubscribedEvents(): array
    {
        return [
            LoginSuccessEvent::class => 'onLoginSuccessEvent',
        ];
    }
}

```

La méthode `getSubscribedEvents` indique bien que l'on souhaite écouter l'événement `LoginSuccessEvent` et exécuter la fonction `onLoginSuccessEvent` lorsque celui-ci est déclenché. En implémentant du code dans la méthode `onLoginSuccessEvent` nous pouvons donc exécuter des actions supplémentaires lorsque l'événement `LoginSuccessEvent` est déclenché.



L'événement `LoginSuccessEvent` provient de la classe `Symfony\Component\Security\Http\Event>LoginSuccessEvent` que nous importons.

Un `eventSubscriber` peut écouter plusieurs événement à la fois, le subscriber créé précédemment pourrait aussi écouter l'événement de déconnexion par exemple. Pour connaître les événements disponible dans l'application il existe la commande suivante :

```
php bin/console debug:event-dispatcher
```

Dans la liste affichée j'ai trouvé deux choses intéressantes ! En rouge l'événement

`LoginSuccessEvent` qui est écouté par 3 fichiers, dont le subscriber `DemoSubscriber` que nous avons créé avec sa méthode `onLoginSuccessEvent` . En vert, l'événement

`LogoutEvent` que nous souhaitons écouter.

```
"Symfony\Component\Security\Http\Event>LoginSuccessEvent" event
-----
Order  Callable                                                                 Priority
-----
#1     App\EventSubscriber\DemoSubscriber::onLoginSuccessEvent()           0
#2     App\EventSubscriber\SecuritySubscriber::onLoginSuccess()             0
#3     Symfony\Component\Security\Http\EventListener>PasswordMigratingListener::onLoginSuccess() 0
-----

"Symfony\Component\Security\Http\Event>LogoutEvent" event
-----
Order  Callable                                                                 Priority
-----
#1     Symfony\Component\Security\Http\EventListener\CsrfTokenClearingLogoutListener::onLogout() 0
-----
```

Pour écouter ce nouvel événement je récupère le chemin du fichier, ici `Symfony\Component\Security\Http\Event\LogoutEvent` qui me met à disposition la classe `LogoutEvent` et je l'indique comme nouvel événement à écouter dans la méthode `getSubscribedEvents` de mon subscriber.

```
<?php

namespace App\EventSubscriber;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Security\Http\Event\LoginSuccessEvent;
use Symfony\Component\Security\Http\Event\LogoutEvent;

class DemoSubscriber implements EventSubscriberInterface
{
    public function onLoginSuccessEvent(LoginSuccessEvent $event): void
    {
        // ...
    }
    public function onLogoutEvent(LoginSuccessEvent $event): void
    {
        // ...
    }

    public static function getSubscribedEvents(): array
    {
        return [
            LoginSuccessEvent::class => 'onLoginSuccessEvent',
            LogoutEvent::class => 'onLogoutEvent',
        ];
    }
}
```

Il ne reste plus qu'à implémenter le code nécessaire dans les différentes fonctions pour personnaliser le comportement de l'application à la connexion / déconnexion de l'utilisateur.

## Écouter les événements Doctrine

 [Documentation Symfony Doctrine events](#)

 [Liste des événements Doctrine disponibles](#)

Tout comme les listener permettant d'écouter les événements Symfony ( exemple, la connexion d'un utilisateur ), il est possible d'écouter les événements de Doctrine. Ceux-ci sont destinés à écouter la création, modification ou encore suppression d'un enregistrement en base de données, lorsque une de ces actions sera exécuté par Doctrine nous pourrons exécuter du code.

Il existe 3 manières d'écouter les événements Doctrine, chacune avec ses avantages et inconvénients :

<b>Lifecycle callbacks</b>	Ils sont définis comme des méthodes publiques sur les classes d'entités	Meilleures performances car ils ne s'appliquent qu'à une seule classe d'entité, mais vous ne pouvez pas réutiliser la logique pour différentes entités et elles n'ont pas accès aux services Symfony
<b>Lifecycle listeners and subscribers</b>	Classes avec des méthodes de rappel pour un ou plusieurs événements et elles sont appelées pour toutes les entités	Ils peuvent réutiliser la logique entre différentes entités et peuvent accéder aux services Symfony, mais leurs performances sont moins bonnes car ils sont appelés pour toutes les entités
<b>Entity listeners</b>	Les écouteurs d'entité, ils sont similaires aux écouteurs lifecycle listeners et subscribers, mais ils ne sont appelés que pour les entités d'une certaine classe	Ils ont les mêmes avantages que les écouteurs de cycle de vie et ils ont de meilleures performances car ils ne s'appliquent qu'à une seule classe d'entité

### Lifecycle callbacks :

Ils sont très utiles pour les actions simples à réaliser sur une entité. Dans le cas où vous devriez réaliser une action sur plusieurs entités, vous devrez répéter votre code sur chaque entité car cette méthode s'applique dans le fichier de l'entité et celui-ci n'a pas accès aux services de Symfony, il sera donc impossible de

mutualiser la logique que vous souhaitez exécuter.

Il peut par exemple être très utile pour mettre à jour automatiquement le champ `created_at` de votre entité à la création d'un `product` :

```
// src/Entity/Product.php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\HasLifecycleCallbacks]
class Product
{
    // ...

    #[ORM\PrePersist]
    public function setCreatedAtValue(): void
    {
        $this->createdAt = new \DateTimeImmutable();
    }
}
```

### Entity listeners :

Ils ont la même vocation que les “lifecycle callbacks” à la différence que ceux-ci nécessitent d’être déclaré dans un fichier PHP à part, de ce fait ils peuvent accéder au service de Symfony. Cependant ils présentent de moins bonnes performances que les “lifecycle callbacks”.

Il peut par exemple être très utile lorsque l’on souhaite envoyer un email à un utilisateur après la création d’un `user` :


```
// src/EventListener/UserChangedNotifier.php

namespace App\EventListener;

use App\Entity\User;
use Doctrine\Bundle\DoctrineBundle\Attribute\AsEntityListener;
use Doctrine\ORM\Events;

#[AsEntityListener(event: Events::postUpdate, method: 'postUpdate', entity: User::class)]
class UserChangedNotifier
{
    public function postUpdate(User $user, PostUpdateEventArgs $event): void
    {
        // ...
    }
}
```

## Mailing

 Pour faciliter le développement de fonctionnalités de mailing en local nous utiliserons Mailhog pour intercepter les mails. Mailhog est un serveur SMTP comme les autres, il est utilisé pour l'envoi et la réception d'email, l'avantage de Mailhog est que nous allons pouvoir utiliser un serveur SMTP provisoire en local.

### Installer Mailhog

```
sudo apt update
sudo apt upgrade

sudo apt-get install golang-go # Mailhog est écrit en go, il est donc nécessaire d'installer go
```



```
mkdir gocode # de préférence à la racine de votre utilisateur
echo "export GOPATH=$HOME/gocode" >> ~/.profile
source ~/.profile
go --version # pour vérifier que l'installation c'est bien déroulée

go get github.com/mailhog/MailHog=
sudo cp /home/<nom_utilisateur>/gocode/bin/MailHog /usr/local/bin/mailhog

~/gocode/bin/MailHog # pour démarrer le serveur Mailhog
```

À la suite de l'installation, exécuter le serveur Mailhog et rendez-vous sur l'url <http://localhost:8025/>

### Configurer Symfony

Vérifier dans le `composer.json` que la librairie Symfony mailer est bien installée, autrement :

```
composer require symfony/messenger symfony/mailer
```

Dans le fichier `.env` à la racine du projet Symfony deux configurations sont à vérifier / modifier

```
...

###> symfony/messenger ###
# Choose one of the transports below
# MESSENGER_TRANSPORT_DSN=amqp://guest:guest@localhost:5672/%2f/messages
# MESSENGER_TRANSPORT_DSN=redis://localhost:6379/messages
MESSENGER_TRANSPORT_DSN=doctrine://default?auto_setup=0
###< symfony/messenger ###
```

```
###> symfony/mailer ###  
MAILER_DSN=smtp://mailhog:1025  
###< symfony/mailer ###
```

### Créer un service d'envoi d'email

Pour ne pas répéter le code inutilement nous allons créer une fonction d'envoi d'email générique, nous ferons appel à celle-ci lorsque nous voudront envoyer un email

Créer un dossier `src/Service` et un fichier `EmailHandler.php`

```
<?php  
  
namespace App\Service;  
  
use Symfony\Bridge\Twig\Mime\TemplatedEmail;  
use Symfony\Component\Mailer\MailerInterface;  
  
class EmailHandler  
{  
    private MailerInterface $mailer;  
  
    public function __construct(  
        MailerInterface $mailer,  
    )  
    {  
        $this->mailer = $mailer;  
    }  
}
```

```

public function sendTemplateEmail(string $email, string $subject, string $template, array $context): void
{
    $email = (new TemplatedEmail())
        ->from('quarter-backend@simplon.com') # paramétrable dans config/packages/mailer.yaml
        ->to($email)
        ->subject($subject)
        ->htmlTemplate($template)
        ->context($context);

    $this->mailer->send($email);
}
}

```

Cette fonction reçoit :

email	string	L'adresse auquel envoyé le mail
subject	string	L'objet de l'email
template	string	Le chemin vers le template Twig utilisé pour l'email, indiquer le chemin depuis le dossier "template"
context	tableau	Tout comme pour les templates de contrôleur vous pouvez passer des variables à votre template d'email

Avant de tester l'envoi d'un email pensez à démarrer le service Messenger de Symfony

```

php bin/console messenger:consume async # mode silencieux
php bin/console messenger:consume async -vv # prompt des événements du mailer

```

Pour tester l'envoi d'un email nous allons créer un contrôleur `EmailController` (supprimable à la suite du test). Ajouter le code suivant dans le contrôleur

⚠ Pensez à créer votre template d'email au préalable

```

<?php

namespace App\Controller;

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

class EmailController extends AbstractController
{
    private MessageHandler $messageHandler;

    public function __construct(MessageHandler $messageHandler)
    {
        $this->messageHandler = $messageHandler;
    }

    #[Route('/email', name: 'app_email')]
    public function index(): Response
    {
        $this->messageHandler->sendTemplateEmail(
            "kevin.wolff@protonmail.com",
            "Email de test",
            "email/emailTemplate.html.twig",
            []
        );

        return $this->render('email/index.html.twig', [
            'controller_name' => 'EmailController',
        ]);
    }
}

```

```
    }  
    }  
};
```

Consulter Mailhog pour voir les mails interceptés par votre server SMTP <http://localhost:8025/>