Formulaire

Ce cours est un résumé depuis : https://symfony.com/doc/current/forms.html

Travailler avec des formulaires HTML est l'une des tâches les plus commune - et prenante - pour un développeur Web. Symfony intègre un composant qui facilite le travail.

Installation

Pour importer le composant formulaire *Symfony Flex*, exécutez la commande suivante:

```
composer require form
```

NOTE : Vous pouvez utiliser le composant formulaire Symphony en dehors de vos projets Symfony. Pour plus d'information, voir *Form component documentation* sur GitHub.

Comment concevoir vos formulaires depuis le controller?

Créer un formulaire avec Symfony demande un peu de code. Créons un nouveau formulaire en utilisant le "form builder" de Symfony.

```
public function new(Request $request)

  {

      // creates a task and gives it some dummy data for this example

      $task = new Task();

      $task->setTask('Write a blog post');

      $task->setDueDate(new \DateTime('tomorrow'));


      $form = $this->createFormBuilder($task)

          ->add('task', TextType::class)

          ->add('dueDate', DateType::class)

          ->add('save', SubmitType::class, array('label' => 'Create Task'))

          ->getForm();
```

```
        return $this->render('default/new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
```

Afficher un formulaire

Maintenant que le formulaire a été créé, il nous reste à l'afficher. Nous pouvons faire cela en passant le formulaire à notre template (remarquez le `$form->createView()` dans le contrôleur ci-dessus) et en utilisant un ensemble de fonctions:

```
{# templates/default/new.html.twig #}

{{ form_start(form) }}

{{ form_widget(form) }}

{{ form_end(form) }}
```

ou

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

Quelques explications :

`form_start(form)`

Permet l'affichage de la balise de début de formulaire, avec le bon enctype lors de l'utilisation de l'upload de fichiers.

`form_widget(form)`

Permet d'afficher tous les champs avec le champs en lui-même, un label et tous les messages de validations ou d'erreurs du champs.

`form_end(form)`

Permet l'affichage de la valise de fin du formulaire et de tous les autres champs qui n'ont pas encore été affiché. Cette option est pratique pour bénéficier de la _Protection CSRF_.

NOTA BENE : `{{ form(form) }}` permet d'afficher les trois block du dessus en une seule fois.

Voir : https://symfony.com/doc/current/forms.html#handling-form-submissions pour la soumissions des formulaires.

Le thème Bootstrap pour les formulaires

Si vous souhaitez appliquer un style bootstrap pour les formulaire, incluez le tag `form_theme` dans les templates où les formulaires sont utilisées:

```
{# ... #}

{# this tag only applies to the forms defined in this template #}

{% form_theme form 'bootstrap_4_layout.html.twig' %}



{% block body %}

 <h1>User Sign Up:</h1>

  {{ form(form) }}

{% endblock %}
```

Symfony affichera votre formulaire en utilisant le layout Bootstrap 4.

Voir plus de formulaire ici : https://symfony.com/doc/current/form/form_customization.html#what-are-form-themes

La documentation complète ici : https://symfony.com/doc/current/forms.html

Sécurité | Symfony 4

Ce cours est un résumé de : https://symfony.com/doc/current/security.html https://symfony.com/doc/current/security/entity_provider.html

Le système de sécurité Symfony est incroyablement performant mais peut être compliqué à mettre en place. Ce chapitre est un résumé et n'aborde que quelques aspects du composant sécurité.

Installation

Utilisez cette commande *Symfony Flex*, pour installer le composant Symfony:

```
composer require security
```

Qu'est-ce que la UserInterface ?

C'est juste une entité simple. Mais pour utiliser cette classe dans le système de sécurité, nous devons implémenter `UserInterface` . Cela force notre classe à avoir les 5 méthodes suivantes :

- getRoles()
- getPassword()
- getSalt()
- getUsername()
- eraseCredentials()

Pour en savoir plus sur chacune de ces fonctions, voyez `UserInterface`.

Que font les méthodes serialize et unserialize ?

A la fin de chaque requête, l'objet user est sérialisé dans la session. A la prochaine requête elle sera désérialisé. Pour aider PHP à faire cela correctement , vous devez implémenter`Serializable`. Mais vous n'avez pas besoin de tout sérialiser: uniquement les champs les plus importants (the ones shown above plus a few extra if you decide to implement AdvancedUserInterface). A chaque requête l' `id` est utilisé pour rafraichir l'objet`User` depuis la base de données.

Rôles

Quand un utilisateur s'authentifie, un rôle lui est attribué (e.g. `ROLE_ADMIN`).

ATTENTION

Tous les rôles que vous assigner à un utilisateur **doivent** commencer par le préfixe`ROLE_`. Sinon, elles ne seront pas prises en compte par le composant sécurité de Symfony.

Les rôles sont simple et sont généralement des chaînes de caractères que vous inventez et utilisez selon vos besoin. Par exemple, si vous souhaitez limiter l'accès administrateur de votre site, vous pouvez protéger cette section en utilisant le `ROLE_BLOG_ADMIN`. Ce rôle n'a pas besoin d'être défini ailleurs, vous pouvez juste l'utiliser.

CONSEIL

Assurez-vous que chaque utilisateur ait au moins *un* rôle. Une des conventions est d'attribuer à *chaque* utilisateur, un `ROLE_USER`.

Vous pouvez aussi spécifier une hiérarchie de rôle.

Hiérarchie de Rôles

Plutôt que d'associer plusieurs rôles aux utilisateurs, nous pouvons définir une hiérarchie :

```
# config/packages/security.yaml

security:

    # ...



    role_hierarchy:

        ROLE_ADMIN:       ROLE_USER

        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

Dans la configuration ci-dessus les utilisateurs avec le `ROLE_ADMIN` ont aussi le `ROLE_USER`. Le `ROLE_SUPER_ADMIN` a aussi `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH` et`ROLE_USER` (hérité de `ROLE_ADMIN`).

Les événements Symfony

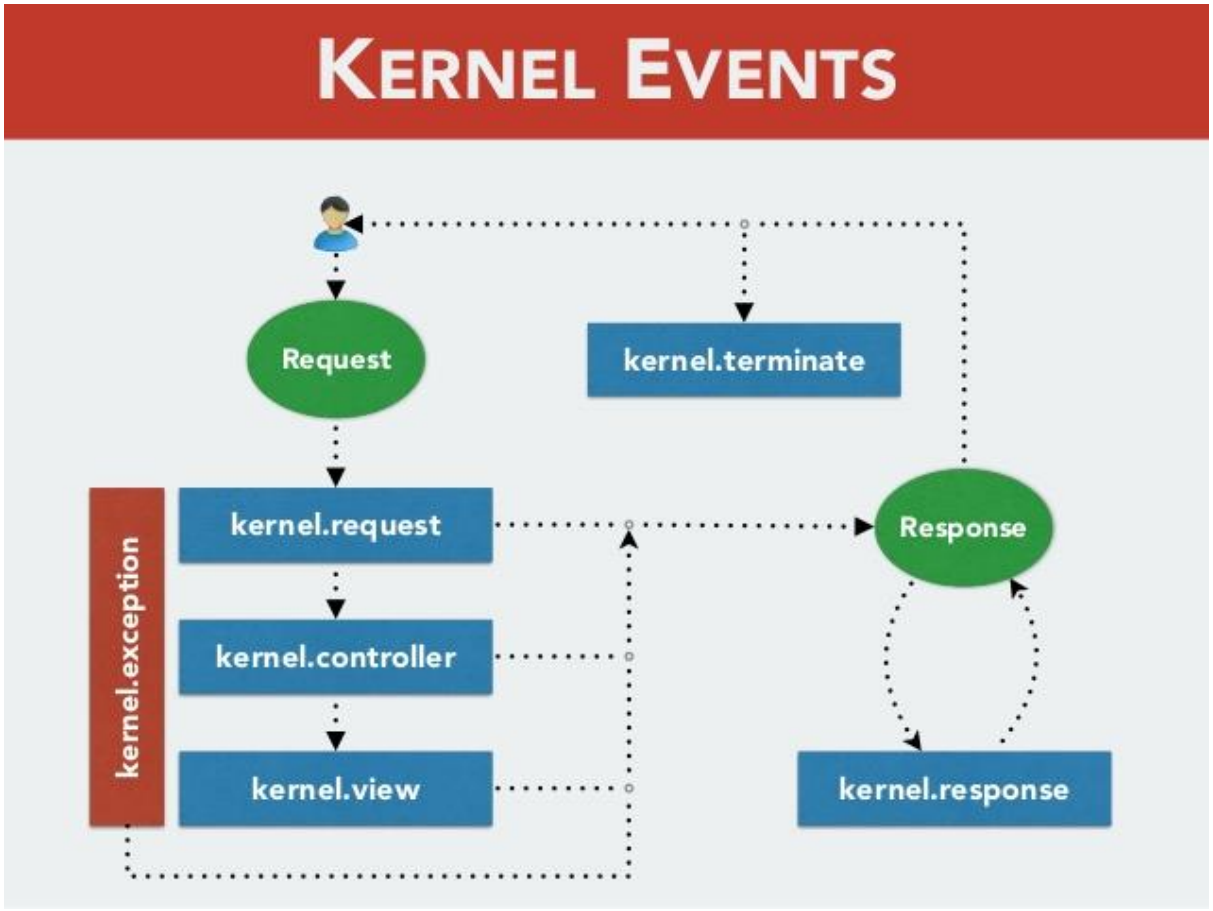
Durant le traitement d'une requête HTTP, le framework Symfony, émet des événements (or any application using the *HttpKernel component*) dispatches some *events* which you can use to modify how the request is handled.

Voir : https://symfony.com/doc/current/reference/events.html

Les événements Kernel de Symfony

Pour bien comprendre, voyons ensemble les événements **kernel** de symfony. Ce ne sont pas les seuls évènements, il en existe d'autres...

MEMENTO : N'oubliez pas, c'est Symfony qui à plusieurs points de son exécution émet des événements. Ici, c'est le Kernel de symfony qui le fait.



- **kernel.request** : C'est le tout premier évènement dispatché par symfony, avant même que le contrôleur ne soit déterminé. Il est utile pour ajouter des informations à la requête ou retourner une réponse avant que la requête ne se fasse.
  https://symfony.com/doc/current/reference/events.html#kernel-request
- **kernel.controller** : Cet évènement est dispatché juste avant l'appel du contrôleur en charge de la requête. *(Le contrôleur n'est donc pas*

encore exécuté). Il est utile pour initialiser des données qui seront utilisés par le contrôleur pour son traitement. https://symfony.com/doc/current/reference/events.html#kernel-controller

- **kernel.view** : Cet évènement est dispatché après l'execution du controleur, mais uniquement si le controleur ne retourne pas de réponse. Il sera donc utile pour transformer la valeur retourné (du HTML par exemple) en une réponse *(Response)* attendu par symfony. https://symfony.com/doc/current/reference/events.html#kernel-view
- **kernel.response** : Cet évènement est dispatché après que le contrôleur est ou qu'un Listener sur `kernel.view` est retournée une réponse `Response`. Cet évènement est utile pour modifier ou remplacer une réponse avant de l'envoyer. *(Par exemple : Modifier les en-tete HTTP, ajouter des cookies, ...)* https://symfony.com/doc/current/reference/events.html#kernel-response
- **kernel.terminate** : Cet évènement est dispatché après qu'une réponse est été exécuté par symfony. Il est pratique pour exécuter certaines tâches qui pourrait ralentir le chargement du site et qui n'ont pas besoin d'être achevés pour envoyer la réponse à l'utilisateur. *(Par exemple, l'envoi de mails...)* https://symfony.com/doc/current/reference/events.html#kernel-terminate

Il existe encore d'autres évènements kernel comme :

- **kernel.exception** : Lorsqu'une erreur se produit. https://symfony.com/doc/current/reference/events.html#kernel-exception
- **kernel.finish-request** : Après une sous-requête. https://symfony.com/doc/current/reference/events.html#kernel-finish-request

Bien-sûr, nous pouvons créer, émettre et écouter nos propres événements !

Les événements sécurité

L'événement `security.interactive_login` est émis après qu'un utilisateur soit connecté sur le site de façon effective:

- authentication based on your session.
- authentication using a HTTP basic header.

Vous pouvez écouter l'événement `security.interactive_login` pour souhaiter la bienvenue avec un message Flash à vos utilisateurs chaque fois qu'ils s'authentifient.