Développeur Web et Web Mobile

Prénom : Fabien
Nom : GIBON

Nom du projet : ECF Hypnos

Lien Github du projet : https://github.com/LuckyMierdy/ECF-Hypnos
URL du site (si vous avez mis votre projet en ligne) : http://blooming-temple-71972.herokuapp.com/ (mais ne fonctionne pas correctement en ligne)

Liste des compétences du référentiel qui sont couvertes par le projet

Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité :

1. Maquetter une application
2. Réaliser une interface utilisateur web statique et adaptable
3. Développer une interface utilisateur web dynamique
4. Réaliser une interface utilisateur avec une solution de gestion de contenu ou ecommerce

Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité

1. Créer une base de données
2. Développer les composants d'accès aux données
3. Développer la partie back-end d'une application web ou web mobile
4. Élaborer et mettre en œuvre des composants dans une application de gestion de contenu ou e-commerce

Résumé du projet en français d'une longueur d'environ 20 lignes soit 200 à 250 mots, ou environ 1200 caractères espaces non compris

Dans le cadre de ma formation chez Studi, j'ai eu à exécuter le développement une application complète, et ce, du 15 mars au 21 avril. Le projet consiste à réaliser une application web/mobile qui permettra de réserver une chambre d'hôtel parmi plusieurs choix d'hôtels.
Imaginons qu'Hypnos est un groupe hôtelier avec un grand patron souhaitant gérer simplement

ces subordonnés qui eux devront gérer leur établissement.
Pour combler cette demande, notre application sera capable de donner le droit admin au patron pouvant ajouter des nouveaux établissements et lier des managers qu'il aura au préalable enregistré. Une fois ces données enregistrées, chaque gérant sera capable de gérer son établissement et d'ajouter les chambres.
Chaque visiteur sera capable de voir un catalogue d'établissement grâce à un clic de découvrir les chambres de l'établissement sélectionné. Afin de pouvoir poser une question, faire une réclamation ou faire une réservation, un utilisateur devra se connecter.

Cahier des charges, expression des besoins, ou spécifications fonctionnelles du projet

1. Un administrateur du groupe Hypnos sera en charge de la gestion des établissements ainsi que de la création et gestion des administrateurs dont le but sera de gérer leur établissement.

2 .Les gérants devront pouvoir créer une chambre dans l'établissement dont il est chargé. Les chambres auront en paramètre : nom, image mise en avant, texte descriptif, prix et galerie d'image.

3. Les visiteurs auront une page "catalogue" ou ils pouront voir les établissements. Si le visiteur le souhaite il pourra alors cliquer sur l'établissement et accédé aux chambres de celui-ci.

4. Une page de réservation permettra aux utilisateurs de réserver une chambre. Si celui-ci n'est pas connecté il sera rediriger vers une page de connexion.

5. Depuis une page profil, l'utilisateur devra pouvoir voir ces réservations et les annuler.

6. Sur la page d'un établissement des boutons réserver seront disponible sous chaque chambre afin de rediriger l'utilisateur sur la page de réservation, les champs établissement et chambre seront prérempli.

7. L'utilisateur devra pouvoir contacter l'établissement depuis une page via un formulaire de contact. Celui-ci réclamera le nom/prénom/email de la personne, un sujet disponible parmi 4 choix et un corps de message.

Spécifications techniques du projet, élaborées par le candidat, y compris pour la sécurité et le web mobile

Voici les différentes technologie que j'ai utilisé pour ce projet :

- Environnement de travail :
  - L'outil de versioning Git me permetant de déposer mon code sur Github après chaque commit.
  - L'IDE Visual Studio Code m'a permis de coder facilement grâce à son auto implementation.
- Partie Frond :
  - Pour développer une application web/mobile j'ai utilisé le language HTML 5 / CSS3 pour la structure.
  - L'application devait être reponsive j'ai donc ajouter bootstrap 5 afin de faciliter ce point.
- Partie Back :
  - Afin d'avoir un code facilement maintenable, j'ai choisi PHP 8.1 et le

framework Symfony pour écrire le back end.

Description de la veille, effectuée par le candidat durant le projet, sur les vulnérabilités de sécurité

Description d'une situation de travail ayant nécessité une recherche, effectuée par le candidat durant le projet, à partir de site anglophone

Afin de faire des fomulaires dans symfony, j'ai été obligé d'aller sur la donumentation technique afin de connaitre les détails technique d'execution. Je me suis ensuite reneigné sur les Form Type afin de pouvoir créer un builder cohérant avec mes besoins.

Extrait du site anglophone, utilisé dans le cadre de la recherche décrite précédemment, accompagné de la traduction en français effectuée par le candidat sans traducteur automatique (environ 750 signes).

Creating and processing HTML forms is hard and repetitive. You need to deal with rendering HTML form fields, validating submitted data, mapping the form data into objects and a lot more. Symfony includes a powerful form feature that provides all these features and many more for truly complex scenarios.

Installation
In applications using Symfony Flex, run this command to install the form feature before using it:
"$composer require symfony/form"

Usage
The recommended workflow when working with Symfony forms is the following:

1. Build the form in a Symfony controller or using a dedicated form class;
2. Render the form in a template so the user can edit and submit it;
3. Process the form to validate the submitted data, transform it into PHP data and do something with it (e.g. persist it in a database).
Each of these steps is explained in detail in the next sections. To make examples easier to follow, all of them assume that you're building a small Todo list application that displays "tasks".

Users create and edit tasks using Symfony forms. Each task is an instance of the following Task class:

```
// src/Entity/Task.php
namespace App\Entity;

class Task
{
   protected $task;
   protected $dueDate;

   public function getTask(): string
   {
      return $this->task;
   }

   public function setTask(string $task): void
   {
      $this->task = $task;
   }

   public function getDueDate(): ?\DateTime
   {
      return $this->dueDate;
   }

   public function setDueDate(?\DateTime $dueDate): void
   {
      $this->dueDate = $dueDate;
   }
}
```

This class is a "plain-old-PHP-object" because, so far, it has nothing to do with Symfony or any other library. It's a normal PHP object that directly solves a problem inside your application (i.e. the need to represent a task in your application). But you can also edit Doctrine entities in the same way.

Form Types

Before creating your first Symfony form, it's important to understand the concept of "form type". In other projects, it's common to differentiate between "forms" and "form fields". In Symfony, all of them are "form types":

a single <input type="text"> form field is a "form type" (e.g. TextType);
a group of several HTML fields used to input a postal address is a "form type" (e.g. PostalAddressType);
an entire <form> with multiple fields to edit a user profile is a "form type" (e.g. UserProfileType).
This may be confusing at first, but it will feel natural to you soon enough. Besides, it simplifies code and makes "composing" and "embedding" form fields much easier to implement.

There are tens of form types provided by Symfony and you can also create your own form types.

Building Forms
Symfony provides a "form builder" object which allows you to describe the form fields using a fluent interface. Later, this builder creates the actual form object used to render and process contents.

If your controller does not extend from AbstractController, you'll need to fetch services in your controller and use the createBuilder() method of the form.factory service.

In this example, you've added two fields to your form - task and dueDate - corresponding to the task and dueDate properties of the Task class. You've also assigned each a form type (e.g. TextType and DateType), represented by its fully qualified class name. Finally, you added a submit button with a custom label for submitting the form to the server.

Creating Form Classes
Symfony recommends putting as little logic as possible in controllers. That's why it's better to move complex forms to dedicated classes instead of defining them in controller actions. Besides, forms defined in classes can be reused in multiple actions and services.

Form classes are form types that implement FormTypeInterface. However, it's better to extend from AbstractType, which already implements the interface and provides some utilities:

```
// src/Form/Type/TaskType.php
namespace App\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('task', TextType::class)
            ->add('dueDate', DateType::class)
            ->add('save', SubmitType::class)
        ;
    }
}
```

The form class contains all the directions needed to create the task form. In controllers extending from the AbstractController, use the createForm() helper (otherwise, use the create() method of the form.factory service):

```
// src/Controller/TaskController.php
namespace App\Controller;

use App\Form\Type\TaskType;
// ...
```

```
class TaskController extends AbstractController
{
    public function new(): Response
    {
        // creates a task object and initializes some data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createForm(TaskType::class, $task);

        // ...
    }
}
```

Every form needs to know the name of the class that holds the underlying data (e.g. App\Entity \Task). Usually, this is just guessed based off of the object passed to the second argument to createForm() (i.e. $task). Later, when you begin embedding forms, this will no longer be sufficient.

So, while not always necessary, it's generally a good idea to explicitly specify the data_class option by adding the following to your form type class:

```
// src/Form/Type/TaskType.php
namespace App\Form\Type;

use App\Entity\Task;
use Symfony\Component\OptionsResolver\OptionsResolver;
// ...

class TaskType extends AbstractType
{
    // ...

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

Traduction :

La création et le traitement de formulaires HTML sont difficiles et répétitifs. Vous devez vous occuper du rendu des champs de formulaire HTML, de la validation des données soumises, du mappage des données du formulaire dans des objets et bien plus encore. Symfony inclut une fonctionnalité de formulaire puissante qui fournit toutes ces fonctionnalités et bien d'autres pour des scénarios vraiment complexes.
Installation

Dans les applications utilisant Symfony Flex, exécutez cette commande pour installer la

fonctionnalité de formulaire avant de l'utiliser :
"composer require symfony/form"

Usage
Le flux de travail recommandé lorsque vous travaillez avec des formulaires Symfony est le suivant :

1. Construisez le formulaire dans un contrôleur Symfony ou en utilisant une classe de formulaire dédiée ;
2. Affichez le formulaire dans un modèle afin que l'utilisateur puisse le modifier et le soumettre.
3. Traitez le formulaire pour valider les données soumises, transformez-le en données PHP et faites quelque chose avec (par exemple, persistez-le dans une base de données).
Chacune de ces étapes est expliquée en détail dans les sections suivantes. Pour rendre les exemples plus faciles à suivre, tous supposent que vous créez une petite application de liste de tâches qui affiche des "tâches".

Les utilisateurs créent et modifient des tâches à l'aide de formulaires Symfony. Chaque tâche est une instance de la classe Task suivante :

```
// src/Entity/Task.php
namespace App\Entity;

class Task
{
   protected $task;
   protected $dueDate;

   public function getTask(): string
   {
      return $this->task;
   }

   public function setTask(string $task): void
   {
      $this->task = $task;
   }

   public function getDueDate(): ?\DateTime
   {
      return $this->dueDate;
   }

   public function setDueDate(?\DateTime $dueDate): void
   {
      $this->dueDate = $dueDate;
   }
}
```

Cette classe est un "plain-old-PHP-object" car, jusqu'à présent, elle n'a rien à voir avec Symfony ou toute autre bibliothèque. C'est un objet PHP normal qui résout directement un problème à l'intérieur de votre application (c'est-à-dire la nécessité de représenter une tâche dans votre application). Mais vous pouvez également modifier les entités Doctrine de la même manière.

Types de formulaire
Avant de créer votre premier formulaire Symfony, il est important de comprendre le concept de "form type". Dans d'autres projets, il est courant de faire la différence entre les "form" et les "form fields". Dans Symfony, tous sont des "form types":

un seul champ de formulaire <input type="text"> est un "form type" (par exemple, TextType) ;
un groupe de plusieurs champs HTML utilisés pour saisir une adresse postale est un « form type » (par exemple PostalAddressType) ;
un <form> entier avec plusieurs champs pour modifier un profil utilisateur est un "form type" (par exemple, UserProfileType).
Cela peut être déroutant au début, mais cela vous semblera naturel assez tôt. En outre, cela simplifie le code et rend la "composition" et "l'intégration" des champs de formulaire beaucoup plus faciles à implémenter.

Il existe des dizaines de types de formulaires fournis par Symfony et vous pouvez également créer vos propres types de formulaires.

Créer des formulaires
Symfony fournit un objet "form builder" qui permet de décrire les champs du formulaire à l'aide d'une interface fluide. Plus tard, ce générateur crée l'objet de formulaire réel utilisé pour rendre et traiter le contenu.

Si votre contrôleur ne s'étend pas à AbstractController, vous devrez récupérer les services dans votre contrôleur et utiliser la méthode createBuilder() du service form.factory.

Dans cet exemple, vous avez ajouté deux champs à votre formulaire - task et dueDate - correspondant aux propriétés task et dueDate de la classe Task. Vous avez également attribué à chacun un type de formulaire (par exemple, TextType et DateType), représenté par son nom de classe complet. Enfin, vous avez ajouté un bouton de soumission avec une étiquette personnalisée pour soumettre le formulaire au serveur.

Création de classes de formulaire
Symfony recommande de mettre le moins de logique possible dans les contrôleurs. C'est pourquoi il est préférable de déplacer les formulaires complexes vers des classes dédiées au lieu de les définir dans les actions du contrôleur. De plus, les formulaires définis dans les classes peuvent être réutilisés dans plusieurs actions et services.

Les classes de formulaire sont des types de formulaire qui implémentent FormTypeInterface. Cependant, il est préférable d'étendre AbstractType, qui implémente déjà l'interface et fournit quelques utilitaires :

```
// src/Form/Type/TaskType.php
namespace App\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType
```

```
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('task', TextType::class)
            ->add('dueDate', DateType::class)
            ->add('save', SubmitType::class)
        ;
    }
}
```

La classe de formulaire contient toutes les instructions nécessaires pour créer le formulaire de tâche. Dans les contrôleurs s'étendant du AbstractController, utilisez l'assistant createForm() (sinon, utilisez la méthode create() du service form.factory) :

```
// src/Controller/TaskController.php
namespace App\Controller;

use App\Form\Type\TaskType;
// ...

class TaskController extends AbstractController
{
    public function new(): Response
    {
        // creates a task object and initializes some data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createForm(TaskType::class, $task);

        // ...
    }
}
```

Chaque formulaire doit connaître le nom de la classe qui contient les données sous-jacentes (par exemple App\Entity\Task). Habituellement, cela est juste deviné en fonction de l'objet passé au deuxième argument de createForm() (c'est-à-dire $task). Plus tard, lorsque vous commencerez à intégrer des formulaires, cela ne suffira plus.

Ainsi, bien que ce ne soit pas toujours nécessaire, il est généralement judicieux de spécifier explicitement l'option data_class en ajoutant ce qui suit à votre classe de type de formulaire :

```
// src/Form/Type/TaskType.php
namespace App\Form\Type;

use App\Entity\Task;
use Symfony\Component\OptionsResolver\OptionsResolver;
// ...

class TaskType extends AbstractType
```

```
{
    // ...

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