Symfony

Créer un nouveau projet Symfony avec Laragon

Après avoir démarré Laragon, cliquer dans le menu : créer un site web rapidemment → Symfony.

Nommer le dossier. Laragon lance alors l'invite de commande et installe symfony/website-skeleton. Mieux vaut être patient, ça prend un certain temps.

Quand c'est fini, on a les messages suivants :

* Run your application:

1. Go to the project directory

2. Create your code repository with the git init command

3. Download the Symfony CLI at https://symfony.com/download to install a development web server

* Read the documentation at https://symfony.com/doc

C'est une bonne idée d'initialiser un dépôt git, c'est sûr.

Pour le 3, c'est juste pas possible parce que le serveur de développement fait maintenant partie du package de base symfony, sauf quand on installe avec Laragon. A la place, on peut utiliser le serveur apache de laragon (de toute façon il tourne) en installant un hôte virtuel.

C'est pas compliqué, dans le dossier du projet, taper l'instruction suivante dans le terminal de commande :

composer require symfony/apache-pack

On a aussi :

* You're ready to send emails.

* If you want to send emails via a supported email provider, install

the corresponding bridge.

For instance, composer require mailgun-mailer for Mailgun.

* If you want to send emails asynchronously:

1. Install the messenger component by running composer require messenger;

2. Add 'Symfony\Component\Mailer\Messenger\SendEmailMessage': amqp to the

config/packages/messenger.yaml file under framework.messenger.routing

and replace amqp with your transport name of choice.

* Read the documentation at https://symfony.com/doc/master/mailer.html

J'ai pas encore creusé ça.

Et pour finir :

* Modify your DATABASE_URL config in .env

* Configure the driver (mysql) and

server_version (5.7) in config/packages/doctrine.yaml

* Write test cases in the tests/ folder

* Run php bin/phpunit

***** NOTE: Now, you can use pretty url for your awesome project :) *****

(Laragon) Project path: C:/laragon/www/nom_projet

(Laragon) Pretty url: http://nom_projet.test

Et on autorise le terminal de commande à apporter des modifications, afin de recharger le serveur apache.

Les trucs sympas à installer en plus :

composer require annotations

Pour pouvoir mettre les routes directement dans le contrôleur :

```
class LuckyController extends AbstractController
{
   /**
    * @Route("/lucky/number")
    */
   public function number(): Response
   {
```

composer require symfony/maker-bundle --dev
Pour créer des commandes, des contrôleurs, des entités (doctrine entity), même des crud…

exemples :

php bin/console make:controller BrandNewController

created: src/Controller/BrandNewController.php
created: templates/brandnew/index.html.twig

Pour les modèles (entities) et la base de donnée :

composer require symfony/orm-pack

On configure la base de données dans le .env en développement. Par défaut, l'URL de la base de données aura le nom du projet avec mysql si on ne le change pas, mettre en commentaire (#) la ligne postgresql et décommenter la ligne mysql.
On devrait ensuite taper : php bin/console doctrine:database:create mais en fait Laragon l'a déjà fait, et c'est même pour ça qu'il y avait déjà le bon nom (à défaut d'avoir déjà le bon serveur de base de données, enlever le #, ça Laragon sait pas faire).

Créer sa première « entité »

Maintenant on peut créer une entité, c'est à dire un modèle (une classe) qui va correspondre à une table de la base de données avec tous ses champs (propriétés de la classe) qui vont correspondre aux colonnes de la table:

php bin/console make:entity

Génère un formulaire dans la console qui demande le nom de l'entité et de créer autant de champs qu'on veut.

Après on a le modèle avec toutes les propriétés (private, comme il se doit) et tous les getters et setters pour ces propriétés (cool!).
Et on n'a plus qu'à taper la commande suivante pour avoir un fichier « migration » qui sera capable de générer la table.

php bin/console make:migration

On effectue la migration :

php bin/console doctrine:migrations:migrate

On peut aller vérifier dans le gestionnaire de base de données, la table y est bien.

Important : on peut rajouter des propriétés dans la classe manuellement, mais on peut aussi refaire make:entity pour mettre à jour la table.
php bin/console make:entity --regenerate
Génère les getters et les setters pour les nouvelles propriétés entrées manuellement.
On refait ensuite la migration : php bin/console doctrine:migrations:migrate pour mettre à jour la table dans la base.

Un contrôleur pour gérer mon entité

php bin/console make:controller nom_de_l'entitéController

Et faire l'implémenation.
Mais comme je suis paresseuse, comme tout développeur qui se respecte, je peux aussi demander à symfony de me faire directement le crud :

php bin/console make:crud nom_de_l'entité

Qui va me générer : le contrôleur, les formulaires, les vues… Evidemment, ça ne veut pas dire que tout le boulot est fait, il va y avoir du boulot côté css pour que ce soit joli à voir, et si mon entité est particulière, il me faudra rajouter du code, par exemple pour télécharger des images.

Formulaire qui télécharge une image

Dans mon crud, il ya tout, sauf la gestion des images. Et comme justement c'est ça qui m'intéresse, c'est pas glop pas glop. J'ai trouvé un petit tutoriel sympa (et en Français) : https://nouvelle-techno.fr/actualites/live-coding-upload-dimages-multiples-avec-symfony-4-et-5

dans services.yaml, parameters, j'ai mis :

images_directory: '%kernel.project_dir%/public/uploads'

Et ajouté l'input 'image' dans le $builder du fichier ImageType, le fichier qui génère automatiquement le formulaire.

```
<?php

namespace App\Form;

use App\Entity\Image;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\FileType;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ImageType extends AbstractType
{
  public function buildForm(FormBuilderInterface $builder, array $o
ptions)
  {
    $builder
      ->add('nom')
      ->add('alt')
      ->add('legend')
      ->add('dossier')
```

```
      ->add('image', FileType::class,[
        'label' => false,
        'multiple' => false,
        'mapped' => false,
        'required' => true])
    ;
  }

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

Et bien sûr, il faut gérer l'image dans le contrôleur également :

```
 public function new(Request $request): Response
  {
    $image = new Image();
    $form = $this->createForm(ImageType::class, $image);
    $form->handleRequest($request);


    if ($form->isSubmitted() && $form->isValid()) {
      $photo = $form->get('image')->getData();
      $nom = $form->get('nom')->getData();
      if ($nom != null)
     { $fichier = $nom.'.'.$photo->guessExtension();}
     else
     {$fichier = $form->get('image')->getData()-
>getClientOriginalName();
      }
        $image->setNom($fichier);

      // On copie le fichier dans le dossier uploads
      $photo->move(
        $this->getParameter('images_directory'),
        $fichier
      );


      $entityManager = $this->getDoctrine()->getManager();
      $entityManager->persist($image);
      $entityManager->flush();

      return $this->redirectToRoute('image_index');
```

```
    }

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

Dans mon cas, je voulais qu'on puisse avoir le choix entre garder le nom d'origine du fichier ou en proposer un nouveau, donc j'ai ajouté

```
  ->add('nom', TextType::class,['required'=>false])
```

et

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
```

dans le fichier ImageType.

Modifier une table

J'ai changé d'avis, je ne veux plus d'un champ « dossier » pour mes images, je veux un champ usage (houlala, non, mot réservé en SQL! « pour » conviendra), c'est le contrôleur qui s'occupera de mettre les images dans le bon dossier selon l'usage.

Je modifie mon entité en conséquence (manuellement), et pour que modifier ma table dans la base :

php bin/console doctrine:migrations:diff

La nouvelle migration contient automatiquement la bonne requête. Je modifie également le champ du formulaire dans ImageType, et partout où c'est nécessaire dans les vues.

Bien évidemment, si le contrôleur doit gérer ses usages, il faut un nombre limité d'usages. Deux en l'occurrence : carrousel (image en deux versions, vignette et grande), et illustration (une seule image de taille moyenne.

```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
class ImageType extends AbstractType
{
  public function buildForm(FormBuilderInterface $builder, array $o
ptions)
  {
    $builder
      ->add('nom', TextType::class,['required'=>false])
      ->add('alt')
      ->add('legend')
       ->add('pour', ChoiceType::class, [ 'choices' => ['carrousel'
=> 'carrousel', 'illustration' => 'illustration']])
```

```
      ->add('image', FileType::class,[
        'label' => 'fichier à télécharger',
        'multiple' => false,
        'mapped' => false,
        'required' => true])
      ->add('vignette', FileType::class,[
        'label' => 'vignette (facultatif)',
        'multiple' => false,
        'mapped' => false,
        'required' => false])
    ;
  }
```

Champs dynamiques dans un formulaire :

Je souhaite qu'en mode édition (modification), on ne puisse pas changer le nom d'une image (le nom correspondant au fichier physique!) et qu'on ne vous propose de télécharger une vignette que si elle manquante et seulement pour les carrousels (les images d'illustration étant en une seule taille). On peut ajouter un évènement de formulaire pour cela.

Dans ImageType, ajouter :

```
use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;
```

Et modifier ainsi la fonction buildform :

```
  public function buildForm(FormBuilderInterface $builder, array $o
ptions)
  {
    $builder
      ->addEventListener(FormEvents::PRE_SET_DATA, function
(FormEvent $event) {
        $image = $event->getData();
        $form = $event->getForm();

        // checks if the Image object is "new"
        // If no data is passed to the form, the data is "null".
        // This should be considered a new "Image"
        if (!$image || null === $image->getId()) {
          $form->add('nom', TextType::class,['required'=>false]);
          $form->add('pour', ChoiceType::class, [ 'choices' => [
'carrousel' => 'carrousel', 'illustration' => 'illustration']]);
          $form->add('image', FileType::class,[
            'label' => 'fichier à télécharger',
            'multiple' => false,
```

```
            'mapped' => false,
            'required' => true]);
          $form->add('vignette', FileType::class,[
            'label' => 'vignette (facultatif, vous pouvez la télécha
rger plus tard)',
            'multiple' => false,
            'mapped' => false,
            'required' => false]);
        }
      });
      $builder
      ->addEventListener(FormEvents::PRE_SET_DATA, function
(FormEvent $event) {
        $image = $event->getData();
        $form = $event->getForm();
        if ($image && $image->getVignette() == false && $image-
>getPour() == "carrousel")
        {
          $form->add('vignette', FileType::class,[
            'label' => 'vignette ',
            'multiple' => false,
            'mapped' => false,
            'required' => false]);
        }
      });

  $builder->add('alt')
      ->add('legend')
    ;
  }
```

Attention, dans le contrôleur, il faudra gérer la possibilité qu'un champ soit ou ne soit pas dans le formulaire.

```
 if (array_key_exists('vignette', $_POST) && $form-
>get('vignette') !== null)
```

$form→get('vignette') !== null : regarde si on a rempli le champ (la valeur), mais employé seul génère une erreur si le champ n'existe pas, c'est array-key-exists('vignette', $_POST) qui va d'abord vérifier l'existence du champ en question.

Les « fixtures » pour ensemencer la base de données

C'est un peu l'équivalent des seeders de Laravel.

```
composer require --dev orm-fixtures
```

Dans le fichier généré src/DataFixtures/AppFixtures.php ajouter : `use App\Entity\Image;`

Voici l'exemple de la docume,tation symfony pour générer des lignes de produits :

```
public function load(ObjectManager $manager)
    {
        // create 20 products! Bam!
        for ($i = 0; $i < 20; $i++) {
            $product = new Product();
            $product->setName('product '.$i);
            $product->setPrice(mt_rand(10, 100));
            $manager->persist($product);
        }

        $manager->flush();
    }
```

Évidemment, je veux des images avec les fichiers correspondants à leur noms dans mon dossier. Faut que je trouve autre chose ! Avec laravel, je m'étais fait un seeder qui récupère les images d'un dossier. Je vais juste l'adapter à Symfony, ça m'évitera de télécharger une à une les images de mon site. Je vais copier les dossiers d'images dans Imageur et lancer cette fonction sur le répertoire petites_images pour avoir mes images de carrousel en base de données, il ne restera plus qu'à personnaliser les alt et légendes, et à rajouter les quelques images isolées.

```
<?php

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;
use App\Entity\Image;

class AppFixtures extends Fixture
{
  public function load(ObjectManager $manager)
  {
    $nb_fichiers = 0;
    if ($dossier = opendir("C:\laragon\www\Imageur\public\uploads\
petites_images"))
    {
      while(false !== ($fichier = readdir($dossier)))
      {
        if($fichier != '.' && $fichier != '..' && $fichier != 
'index.php')
        {
          $nb_fichiers++;
```

```
            $image = new Image();
            $image->setNom($fichier);
            $image->setAlt('à compléter!');
            $image->setLegend('');
            $image->setPour('carrousel');
            $image->setVignette(true);
            $manager->persist($image);

          }

        }
        $manager->flush();
      }
      else { dd('pas bon le chemin');}
    }
}
```

Essayons :

`php bin/console doctrine:fixtures:load`

Par défaut cette commande efface ce qu'il y avait précédemment dans la base, si on ne veut pas faut ajouter `--append`.

Ouille, il reste quelques images avec accents dans les noms de fichiers, pas glop pas glop !

Une fois ce problème corrigé, ça marche ! 233 images rentrées dans la base d'un coup !

Relations entre les entités

Détails sur https://sites.google.com/site/symfonikhal/p3-gerer-base-de-donnees-avec-doctrine/3-les-relations-entre-entites

J'ai créé une entité Slider pour mes carrousels, et j'ai besoin d'une relation many-to-many entre les carrousels et les images (un carrousel a évidemment plusieurs images, mais une image peut appartenir à plusieurs carrousels, par exemple une image « alzbrulepangarecrinslaves.jpg » peut appartenir aux carrousels « alezan », « sooty », « pangaré » et « crinslavés »).

Dans mon entité Slider, qui va être l'entité propriétaire de la relation, je crée une propriété images :

```
/**
 * @ORM\ManyToMany(targetEntity="App\Entity\Image", cascade={"persist"})
 */
 private $images;
```

Avec *php bin/console make:entity --regenerate*

Symfony va me créeer un constructeur pour le tableau de collection images, un getter, et non pas un setter mais deux, un qui ajoute une image, un qui enlève une image.

```
public function __construct()
 {
   $this->images = new ArrayCollection();
 }

/**
  * @return Collection|Image[]
  */
  public function getImages(): Collection
  {
    return $this->images;
  }

  public function addImage(Image $image): self
  {
    if (!$this->images->contains($image)) {
      $this->images[] = $image;
    }

    return $this;
  }

  public function removeImage(Image $image): self
  {
    $this->images->removeElement($image);

    return $this;
  }
}
```

Pour que Symfony créée la table dans la base de données :

```
php bin/console doctrine:schema:update –dump-sql (pour voir)
php bin/console doctrine:schema:update –force (pour faire)
```

Dans le contrôleur de Slider, on envoie la collection d' images comme options du formulaire (à la création et à l'édition) :

```
use App\Entity\Image;

//....

  /**
  * @Route("/new", name="slider_new", methods={"GET","POST"})
```

```
   */
  public function new(Request $request, ImageRepository $imageRepos
itory): Response
  {
    $slider = new Slider();

    $images = $imageRepository->findAll();

    $form = $this->createForm(SliderType::class, $slider, [
'images'=> $images]);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
      $entityManager = $this->getDoctrine()->getManager();
      $entityManager->persist($slider);
      $entityManager->flush();

      $images = $form->get('images')->getData();
      foreach ( $images as $image)
      {
        $image = $imageRepository->find($image);
        $slider->addImage($image);
      }
  $entityManager->flush();


      return $this->redirectToRoute('slider_index');
    }

    return $this->render('slider/new.html.twig', [
      'slider' => $slider,
      'form' => $form->createView(),
    ]);
  }
 /**
   * @Route("/{id}/edit", name="slider_edit", methods={"GET","POST"
})
   */
  public function edit(Request $request, Slider $slider, ImageRepos
itory $imageRepository): Response
  {
    $images = $imageRepository->findAll();

    $form = $this->createForm(SliderType::class, $slider, [
'images'=> $images]);
    $form->handleRequest($request);
```

```
    if ($form->isSubmitted() && $form->isValid()) {
      $this->getDoctrine()->getManager()->flush();

      $images = $form->get('images')->getData();
      foreach ( $images as $image)
      {
        $image = $imageRepository->find($image);
        $slider->addImage($image);
      }
 $this->getDoctrine()->getManager()->flush();


      return $this->redirectToRoute('slider_index');
    }

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

Dans le fichier SliderType :

```
<?php

namespace App\Form;

use App\Entity\Slider;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use App\Entity\Image;


class SliderType extends AbstractType
{
  public function buildForm(FormBuilderInterface $builder, array $o
ptions)
  {
    $photos = [];

    foreach ($options['images'] as $image)
    {
      $photos[$image->getNom()]  = $image->getId(); //on affiche le
nom, on transmet l'id!
```

```
    }

    $builder
      ->add('nom')
      ->add('article')
      ->add('section')
      ->add('images', ChoiceType::class,[
        'choices' => $photos,
        'multiple' => true,
        'mapped' => false,
        'required' => false])
    ;
  }

  public function configureOptions(OptionsResolver $resolver)
  {
    $resolver->setDefaults([
      'data_class' => Slider::class,
      'images' => [],

    ]);

  }
}
```

Quelques améliorations

Finalement, je sors les fichiers images du formulaire parce que des images sans images, c'était pas facile à exploiter, et je leur fais deux formulaires dédiés ou apparaissent pour l'un les images disponibles (qui ne sont pas dans le carrousel) avec des cases à cocher pour les ajouter en nombre, et pour l'autre les images déjà insérées, avec des cases à cocher pour les enlever.

Générer les carrousels PHP

Dans mon fichier yaml, j'indique l'emplacement du dossier où ranger mes fichiers générés, comme je l'avais fait pour les images téléchargées.

```
parameters:
  images_directory: '%kernel.project_dir%/public/uploads'
  generated_directory: '%kernel.project_dir%/public/generated'
```

On va utiliser ce paramètre dans le contrôleur (SliderController) lors de l'appel de la fonction qui génère le carrousel :

```
 public function sliderGenere(Slider $slider)
  {
```

```
    $dir = $this->getParameter('generated_directory');
    $slider->genereSlider($dir);

    return $this->redirectToRoute('slider_index');
  }
```

La fonction qui va générer le fichier est une méthode de la classe Slider :

```
public function genereSlider($dir)
  {

    $path = $dir."/slider_".$this->getNom().".php";
    $sliderFile = fopen($path, 'w');

    fwrite($sliderFile,'<div class="container"> ');
    foreach ($this->getImages() as $image)
    {
      fwrite($sliderFile, '<figure class="slide"> ');

      fwrite($sliderFile, ' <img class="clickable" src="petites_ima
ges/'.$image->getNom().'" width=150 height=100 onclick="displaySlid
es(src) ;"> ');
      fwrite($sliderFile, '<figcaption hidden>'.$image-
>getLegend().'</figcaption>');
      fwrite($sliderFile, '</figure> ');
    }

    fwrite($sliderFile,'</div>');
  }
```