

Symfony

Créer un nouveau projet Symfony avec Laragon

Après avoir démarré Laragon, cliquer dans le menu : créer un site web rapidement → 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'implémentation.

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 $options)
```

```
    {
```

```
        $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, 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.