

## CRUD Symfony

Lancer le serveur : `php -S localhost:8000 -t public`

On pourra initialiser un nouveau projet, je vous laisse faire cela seul, désormais vous connaissez la marche à suivre et la doc est assez clair à ce sujet (pensez à enlever la version).

On peut vérifier avec la commande `php bin/console` afin de vérifier que vous avez bien accès à la catégorie make vous savez à quel point elle est importante maintenant, si oui on est prêt sinon `composer req maker`, et on sera prêt à dev notre appli.

On va pouvoir créer notre **Bdd** que l'on appellera **Crud** pour ce mini projet ou comme vous le voulez si vous préférez autre chose, cela se passe dans le fichier `.env`, `php bin/console doctrine:database:create`, on fera la table plus tard pas de panique. On va créer notre premier **Controller** je vais faire simple et l'appeler **MainController**, voilà le **Controller** et le **twig** sont générés, vous pouvez ouvrir votre première page sur le navigateur.

On va créer notre **entity** je vais encore faire simple et l'appeler **crud** tout bêtement, on créera plusieurs field :

- title (string 255 not null)
- content(string 255 not null)

Ce sera avec la commande `php bin/console make:entity`, cette commande vous demandera le nom de l'entité et les champs donc vous les avez plus haut.

Ce sera suffisant pour le mini projet du crud, vous pouvez effectuer la migration `php bin/console make:migration`, ensuite vous pouvez faire la migration vers votre Bdd `php bin/console doctrine:migrations:migrate`, et là, votre **Bdd** ainsi que la **table** sera visible dans php myAdmin. On est prêt à coder désormais, let's go.

En résumé :

- Configurer `.env`
- Crée la base : `php bin/console doctrine:database:create`
- Crée l'entité : `php bin/console make:entity`
- Crée la migration : `php bin/console make:migration`
- Exécute la migration : `php bin/console doctrine:migrations:migrate`

On va créer notre formulaire car il faut bien un formulaire pour créer nos données en Bdd, `php bin/console make:form` et on l'appellera **CrudType** simplement, vous aurez une question vous demandant sur quel **entity** le **form** sera relié donc pour nous ce sera **Crud**, sinon regarder le nom dans votre dossier **entity**. Vous avez désormais un dossier **Form** qui a été créé ainsi que votre formulaire **CrudType**. On va aller installer **bootstrap** pour nous faciliter la tâche sur le design, vous savez faire désormais. Allons modifier certaine chose dans notre **controller**, on va définir le chemin d'origine en page d'accueil (nouveau truc eheh), et créer un chemin pour notre form ainsi que quelques précisions.



```

/**
 * @Route("/", name="app_main")
 */
public function index(): Response
{
    return $this->render('main/index.html.twig', [
        'controller_name' => 'MainController',
    ]);
}

```

Ensuite on crée la route pour le form,

```

/**
 * @Route("/create", name="app_main")
 */
public function create(Request $request): Response
{
    $crud = new Crud();
    $form = $this->createForm(CrudType::class, $crud);
    $form->handleRequest($request);

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

```

Voyons ces lignes ensemble :

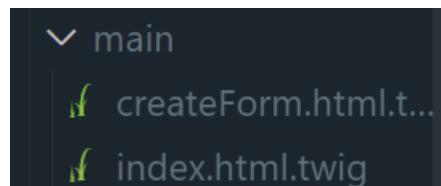
- \$crud = new Crud(); veut dire une nouvelle instance de la classe crud ça vous connaissez
- \$form = \$this->createForm(CrudType::class, \$crud); veut dire on crée un formulaire Symfony en utilisant la classe **CrudType**. Le formulaire est associé à l'instance **\$crud** créée à la ligne précédente. La méthode **createForm()** est une méthode fournie par Symfony pour créer des formulaires.
- \$form->handleRequest(\$request); veut dire que l'on traite la requête http actuelle(**\$request**) et l'applique au formulaire créé à la ligne précédente. Ça veut dire que symfony va analyser les données soumises dans la requête et les appliquer aux champs du formulaire, si la requête est une requête de soumission, Symfony va envoyer les données à **\$crud**, si jamais la requête est une requête de soumission avec des erreurs de validation et bien Symfony va renseigner les erreurs dans le formulaire

On va modifier une dernière petite chose encore, on va modifier le rendue **twig** et créer le nouveau fichier à l'intérieur du dossier main.

```

return $this->render('main/createForm.html.twig', [
    'controller_name' => 'MainController',
    'form' => $form->createView()
]);

```



Voilà, on est tranquille maintenant. On peut s'attaquer à la création du formulaire, il sera basique soyons clair.

```

<div class="container">
    <div class="row">
        <h3 class="mt-5 mb-5">Formulaire de création</h3>
        <div class="col-md-6 offset-md-3">
            {{ form_start(form) }}
            {{ form_widget(form) }}
            {{ form_end(form) }}
        </div>
    </div>
</div>

```

Ici nous précisons le début et la fin du formulaire, ainsi que l'affichage du formulaire avec les champs décidés précédemment, donc le **title** et le **content**. Ajoutons un peu de style à tout ça.

```

{{ form_start(form) }}
    <div class="form-group">
        {{ form_label(form.title) }}
        {{ form_widget(form.title, { 'attr': {'class': 'form-control'} }) }}
    </div>
    <div class="form-group">
        {{ form_label(form.content) }}
        {{ form_widget(form.content, { 'attr': {'class': 'form-control'} }) }}
    </div>
    <div class="form-group">
        <button class="btn btn-info mt-2 text-white">Soumettre</button>
    </div>
{{ form_end(form) }}

```

Voilà un formulaire personnalisé et stylisé vous verrez tout de suite la différence 😊, voilà nous en avons fini avec notre formulaire côté rendu, on doit passer à la gestion du formulaire dans notre **Controller**.

```

public function create(Request $request): Response
{
    $crud = new Crud();
    $form = $this->createForm(CrudType::class, $crud);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $sendDatabase = $this->getDoctrine()
            ->getManager();
        $sendDatabase->persist($crud);
        $sendDatabase->flush();

        $this->addFlash('notice', 'Soumission réussi !!');
    }
}

```

Ici nous avons géré l'envoi du formulaire en Bdd si le formulaire est soumis et valid, vous connaissez désormais le **getDoctrine** ainsi que le **getManager**, le **persist** et le **flush** et oui on parle le langage dev et vous comprenez eh eh, ensuite nous avons géré l'apparition d'un message **flash** qui confirme l'envoie des données en Bdd, c'est un indication que j'aime donné aux utilisateurs pour confirmer la réussite, le '**notice**' permet de définir différent message selon l'action réalisé, comme un composant que l'on pourrait réutiliser mais en changeant les paramètres donc le message. Allons afficher cela dans notre **twig**,

```

<div class="row">
    <h3 class="mt-5 mb-5">Formulaire de création</h3>
    {# Affichage d'un message flash #}
    {% for message in app.flashes('notice') %}
        <div class="alert alert-success">
            {{message}}
        </div>
    {% endfor %}
    {# Affichage d'un formulaire Symfony #}
    <div class="col-md-6 offset-md-3">

```

Voilà qui est mieux, et bien allons tester cela de suite en envoyant notre premier message en **Bdd**, et l'affichage du message flash. Parfait tout fonctionne à merveille. Nous avons donc pour le moment le **create** du **CRUD**. Nous allons gérer l'affichage des messages donc le **read**, vers notre page d'accueil, et s'occuper de l'**update** et du **delete**, et nous aurons vu tout le **CRUD**. Nous allons gérer la redirection également vers la page d'accueil après validation du formulaire. Retournons sur notre **Controller** pour gérer cela.

```

$this->addFlash('notice', 'Soumission réussi !!');

return $this->redirectToRoute('main');

```

Et voilà la redirection est déjà gérée, et oui c'est aussi simple que cela, vous pouvez tester de suite. Parfait encore une fois ça fonctionne à merveille. Allons chercher une table dans **Bootstrap** pour afficher cela plus proprement, j'ai pris la première tout simplement, on l'affiche et on retourne sur le **Controller**, de manière à pouvoir afficher toute nos données venant du **repo** et de l'**entity**. A vous de jouer !!!

```
/*
 * @Route("/", name="main")
 */
public function index(): Response
{
    $data = $this->getDoctrine()->getRepository(Crud::class)->findAll();
    return $this->render('main/index.html.twig', [
        'controller_name' => 'MainController',
        'data'=>$data,
    ]);
}
```

On retourne sur notre affichage du **twig** et on affiche nos données, Essayer seul aussi !!!

```
<tr>
    <th scope="col">#</th>
    <th scope="col">Title</th>
    <th scope="col">Content</th>
    <th scope="col">Action</th>
</tr>
</thead>
<tbody>
    {% for data in datas %}
        <tr>
            <th scope="row">{{data.id}}</th>
            <td>{{data.title}}</td>
            <td>{{data.content}}</td>
            <td>
                <a href="{{ path('update',{'id': data.id}) }}" class="btn btn-warning">Modifier</a>
                <a href="{{ path('delete',{'id': data.id}) }}" class="btn btn-danger">Supprimer</a>
            </td>
        </tr>
    {% endfor %}

```

On doit désormais créer nos routes pour **update** et **delete** bien évidemment dans notre Controller, commençons par le **update**,

```

    /**
 * @Route("/update/{id}", name="update")
 */
public function update(Request $request, $id): Response
{

    $crud = $this->getDoctrine()->getRepository(Crud::class)->find($id);
    $form = $this->createForm(CrudType::class, $crud);
    $form->handleRequest($request);
    if ( $form->isSubmitted() && $form->isValid() ) {
        $sendDatabase = $this->getDoctrine()
                        ->getManager();
        $sendDatabase->persist($crud);
        $sendDatabase->flush();

        $this->addFlash('notice', 'Modification réussie !!');

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

```

```

return $this->render('main/updateForm.html.twig', [
    'controller_name' => 'MainController',
    'form' => $form->createView()
]);

```

Voici pour l'**update** vous noterez qu'il n'y a que très peu de différence avec le **create**, ici il n'est plus question d'instancier la classe **Crud** mais bien de récupérer les données du **crud/form** par son id et de pouvoir le mettre à jour, à part cela et le message qui change évidemment il n'y a rien d'autre à modifier, on peut donc passer au **delete** et nous aurons fini, il ne nous restera ensuite plus qu'à tester tout cela. Voyons le **twig** de l'**update**, qui lui aussi ne va pas beaucoup changer, on pourra copier le formulaire entier du **create** et modifier le **titre** et le **bouton**, passons au **delete** dans notre **Controller**, qui lui sera encore plus facile à gérer comme nous avons pensé à tout jusqu'à présent.

```

    /**
 * @Route("/delete/{id}", name="delete")
 */
public function delete($id): Response
{

    $crud = $this->getDoctrine()->getRepository(Crud::class)->find($id);
    $sendDatabase = $this->getDoctrine()
                        ->getManager();
    $sendDatabase->remove($crud);
    $sendDatabase->flush();

    $this->addFlash('notice', 'Suppression réussie !!');

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

```

Ici nous avons enlevé bien sur la **création du form** car inutile dans le cas du **delete**, nous avons changé la méthode **persist** en **remove** cela est plutôt logique on ne veut pas ajouter dans données en **Bdd** mais en enlever, la modification du message flash et c'est tout, on à enlever la **redirection** car ce n'est pas utile également, et bien voilà nous avons vu le **CRUD** en totalité, vous savez désormais faire les 4 étapes du **CRUD**, la **création**, la **lecture**, la **modification** et la **suppression** de vos données, bravo à vous, vous êtes prêt à commencer de plus gros projet avec un espace connexion pour vos membres et même plus encore, la seule limite sera votre imagination. Je vous conseille de regarder dans la doc Symfony **UserPasswordHasherInterface**, qui permet de hacher le mot de passe de vos **users** donc on **sécurise**, ainsi que la partie **Security** qui se trouve sur le bas de la page de la doc, vous y trouverez tout ce dont vous avez besoin pour sécuriser votre application avec des exemple bien précis ainsi que les lignes de commande pour installer ce dont vous aurez besoin, ainsi que la partie **Errors/Debugging** dans **The basics**. Vous serez paré à tout éventualité, et toutes les attaques les plus fréquentes Symfony gère cela d'une excellent manière et est assez simple à mettre en place c'est pourquoi je vous laisse voir cela seul, la doc doit devenir un réflexe pour vous car cela restera votre plus fidèle allié tout au long de votre carrière.