

## AP3.2 : Un projet PHP avec le Framework Symfony



# Jardinier

### PRÉSENTATION

Présentation de l'équipe et du projet	
Durée du projet	Du 17/12/2021 au 30/03/2022

### OBJECTIFS

#### Découverte de la mission et ses objectifs

- **Environnement de travail**
- **Gestion des devis** : créer un devis : choix parmi différentes haies, saisi des dimensions : crée une facture avec le détail des prix.
- **Gestion des haies et catégories** : recherche/ajout/modification/suppression
- **Gestion des clients** : recherche/ajout/modification/suppression des clients



#### **Bonus**

→ [Système d'espace membre](#)



## → Système de gestion des erreurs

## Captures Ecrans

The screenshot shows the 'Crée un devis' form. On the left is a sidebar with navigation links: Accueil, Devis, Haies, Créé haie, Voir les haies, and Catégories. The main area has a header 'Jardinier' with user and power icons. Below the header is a large orange circle icon with a document and a green '55' badge. The form title is 'Crée un devis'. It contains a dropdown menu labeled '--Choisir votre type de haie--', two input fields for 'Longueur en cm' and 'Hauteur en cm', and a blue button 'Ajouter la haie au devis'.

The screenshot shows the 'Devis' view. The sidebar is the same as the previous screenshot. The main area has a header 'Jardinier' with user and power icons. Below the header is a large green circle icon with a person and a green '55' badge. The form title is 'Devis'. It contains a header 'Jardinier' with a logo and the text 'Crée le : 30/03/2022'. Below this is a table with columns: Type, Prix unitaire, Longueur, and Hauteur. The table has one row: 'Charloote', '3,00 €', '33 cm', and '45 cm'. To the right of the table is a button 'X'. Below the table is a button 'Ajouter la haie au devis'.

The screenshot shows the 'Les types de haies' view. The sidebar is the same as the previous screenshots. The main area has a header 'Jardinier' with user and power icons. Below the header is a large green circle icon with a person and a green '55' badge. The form title is 'Les types de haies'. It contains a grid of four items, each with a green circle icon, a name, a price, and a button 'X'.

Type	Prix unitaire	Longueur	Hauteur
Charloote	3,00 €	33 cm	45 cm

Total: 99,00 €

## Conclusion

Ce projet m'a permis de découvrir **un nouveau framework MVC** plus complet afin de gérer au mieux les différents projets. La mise en place de Bootstrap a permis au projet d'avoir un aspect esthétique moderne et flexible.

# Création/Suppression/Modification des haies et Catégories

Nous avons maintenant un modèle objet persistant opérationnel. On peut créer, afficher, modifier et supprimer des objets de ce type.

Les contrôleurs associés à ces données permettront de lister toutes les données des tables, de visualiser une donnée en fonction de son id et d'éditer les données.

Par exemple, pour la table Haie, les routes suivantes sont attendues :

- /haie
- /haie/{code}
- /haie/creer/
- /haie/modifier/{code}
- /haie/supprimer/{code}

## Créer un objet : création d'un enregistrement dans la table

- Au début du contrôleur, après l'instruction namespace, ajouter l'instruction suivante :

```
use App\Entity\nomClasseMetier;
```

Dans le cas de notre application, on rajoutera : `use App\Entity\Haie;`

## Le principe

- l'accès au gestionnaire d'entités (entity manager) via :  
`$this->getDoctrine()-getManager()`
- c'est l'objet qui fait réellement les requêtes
- la méthode `persist` qui indique à l'entity manager que l'objet passé en paramètre doit être persisté
- la méthode `flush` exécute toutes les requêtes nécessaires en un seul `prepare`.
- on peut faire plusieurs `persist` pour un seul `flush` (plusieurs objets 'persister' avant de faire le `flush`).

```
class HaieController extends AbstractController
{
    /**
     * @Route("/haie/creer", name="creer_haie")
     */
    public function creer_haie(): Response
    {
        $entityManager = $this->getDoctrine()-getManager();

        $haie = new Haie();
        $haie->setCode('LA');
        $haie->setNom('Laurier');
        $haie->setPrix(30);
    }
}
```

```
$entityManager->persist($haie);
$entityManager->flush();
return new Response('Type de haie créé avec le code
' . $haie->getCode());
}
/**
 * @Route("/haie", name="haie")
 */
public function index(): Response
{
    return $this->render('haie/index.html.twig', [
        'controller_name' => 'HaieController',
    ]);
}
```



### Création une catégorie

Nom de la catégorie

Ajouter la catégorie

## Les types de haies



Charlotte



Nugats



# Gestion devis

J'ai utilisé :

```
php bin/console make:entity
```


Crée un devis

Choisir votre type de haie

Longueur en cm

Hauteur en cm

Ajouter la haie au devis

 **Jardinier**

Crée le : 30/03/2022

Jardinier, Inc.  
Rue de la Jardinierie  
JardinCity, CP 12345

Vous êtes un Particulier

Type	Prix unitaire	Longueur	Hauteur	
Charloote	3,00 €	33 cm	44 cm	X

Total: 99,00 €

← +

Doctrine va créer un objet Haie, et voici la méthode qu'il utilisé pour faire le lien entre deux entités.

```
/**
 * @ORM\ManyToOne(targetEntity=Categorie::class, inversedBy="haies")
 * @ORM\JoinColumn(nullable=false)
 */
private $categorie;

public function getCategorie(): ?Categorie
{
    return $this->categorie;
}

public function setCategorie(?Categorie $categorie): self
{
    $this->categorie = $categorie;
}
```

```
        return $this;  
    }  
}
```

Ensuite il nous suffit de migrer nos données vers la BDD

```
php bin/console make:migration  
php bin/console doctrine:migrations:migrate
```

Il faut également ajouter les instructions d'importations des classes correspondant aux types de contrôles graphiques allant être utilisés sur le formulaire :

```
use Symfony\Component\Form\Extension\Core\Type\TextType;  
use Symfony\Component\Form\Extension\Core\Type\NumberType;  
use Symfony\Component\Form\Extension\Core\Type\DateType;  
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
```

- Les différents types utilisés pour créer le formulaire :  
<https://symfony.com/doc/current/reference/forms/types.html>
- La méthode `createView()` du formulaire pour la création du template

```
class CreerHaieController extends AbstractController  
{  
    /**  
     * @Route("/creer/haie", name="creer_haie")  
     */  
    public function index(Request $request): Response  
    {  
        $haie = new Haie();  
        $form = $this->createFormBuilder($haie)  
            ->add('code', TextType::class, array('label'=>'Code haie'))  
            ->add('nom', TextType::class, array('label'=>'Nom haie'))  
            ->add('prix', NumberType::class, array('label'=>'Tarif haie',  
'invalid_message'=>'Saisir un nombre'))  
            ->add('save', SubmitType::class, array('label' => 'AJOUTER'))  
            ->getForm();  
  
        return $this->render('creer_haie/index.html.twig',  
            array('form' => $form->createView())  
        );  
    }  
}
```

## La vue

La vue doit d'abord hériter du layout défini plus haut (par une instruction `extends`), puis redéfinir le bloc content. Dans ce dernier, on va afficher le formulaire, contenu dans la variable `form` transmise par le formulaire, ou afficher un message informant que la saisie a bien été effectuée.

```
{% extends 'base.html.twig' %}

{% block title %}Hello CreerHaieController!{% endblock %}

{% block body %}
    {% block content %}
        <h2>Ajouter un type de haie</h2>
        {{ form(form) }}
    {% endblock %}
{% endblock %}
```

## Gérer une requête de formulaire

On a affiché un formulaire à partir d'un objet existant. Maintenant on veut récupérer les données venant de l'utilisateur pour créer ou modifier des entités.

- La méthode `handleRequest()` de la classe `Form` permet de récupérer les valeurs des champs dans les inputs du formulaire.
- La méthode `isSubmitted()` de la classe `Form` permet de savoir si on est effectivement en méthode POST.
- La méthode `isValid()` permet de valider les données saisies.

On note que la méthode `handleRequest` est toujours appelée avant la méthode `createView` du formulaire. Ceci pour permettre l'affichage des erreurs de validation dans la vue.

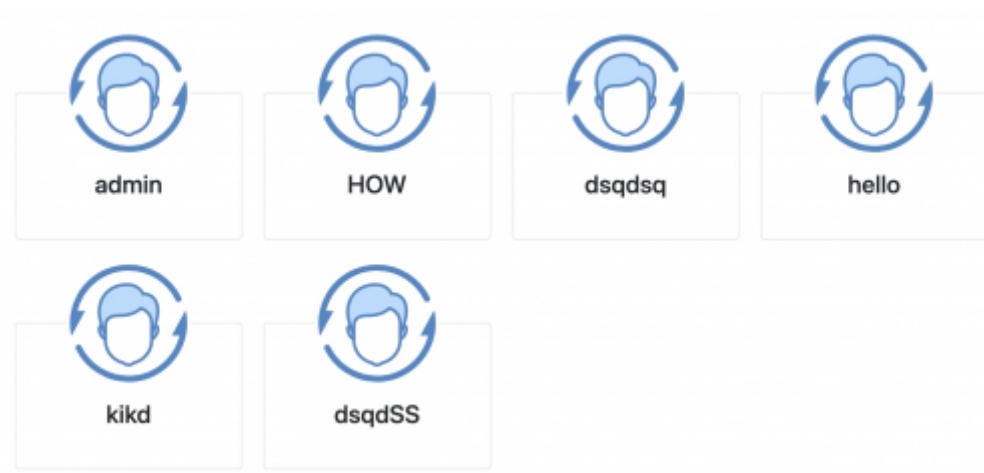
```
$form->handleRequest($request);
if ($form->isSubmitted() && $form->isValid()) {
    // A compléter
}
return $this->render('creer_haie/index.html.twig',
    array('form' => $form->createView())
);
```



# Gestion utilisateur/client

J'ai utilisé :

```
php bin/console make:entity
```



A screenshot of a user profile form. At the top, there is a blue circular icon with a white silhouette of a person's head and shoulders. Below the icon, the text "Votre profil" is displayed. The form contains several input fields: "Particulier", "dsqdqs", and "dsqdqs". Below these fields, the section "Lieu" is visible, containing three input fields: "Adresse" (with the value "dsqdqsdqs"), "Ville" (with the value "dsqdqs"), and "Code Postal" (with the value "33333"). At the bottom of the form, there is a blue button labeled "Mettre à jour".



# Jardinier

Doctrine va créer un objet Utilisateur, et voici la méthode qu'il utilise pour faire le lien entre deux entités.

```
/**
 * @ORM\ManyToOne(targetEntity=Utilisateur::class,
inversedBy="utilisateur")
 * @ORM\JoinColumn(nullable=false)
 */
private $categorie;

public function getCategorie(): ?Categorie
{
    return $this->categorie;
}

public function setCategorie(?Categorie $categorie): self
{
    $this->categorie = $categorie;

    return $this;
}
```

Ensuite il nous suffit de migrer nos données vers la BDD

```
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

Il faut également ajouter les instructions d'importations des classes correspondant aux types de contrôles graphiques allant être utilisés sur le formulaire :

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\NumberType;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
```

- Les différents types utilisés pour créer le formulaire :  
<https://symfony.com/doc/current/reference/forms/types.html>
- La méthode `createView()` du formulaire pour la création du template

```
class CreerHaieController extends AbstractController
{
    /**
     * @Route("/creer/haie", name="creer_haie")
     */
    public function index(Request $request): Response
    {
        $haie = new Haie();
        $form = $this->createFormBuilder($haie)
            ->add('code', TextType::class, array('label'=>'Code haie'))
            ->add('nom', TextType::class, array('label'=>'Nom haie'))
            ->add('prix', NumberType::class, array('label'=>'Tarif haie',
            'invalid_message'=>'Saisir un nombre'))
            ->add('save', SubmitType::class, array('label' => 'AJOUTER'))
            ->getForm();

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

## La vue

La vue doit d'abord hériter du layout défini plus haut (par une instruction `extends`), puis redéfinir le bloc `content`. Dans ce dernier, on va afficher le formulaire, contenu dans la variable `form` transmise par le formulaire, ou afficher un message informant que la saisie a bien été effectuée.

```
{% extends 'base.html.twig' %}

{% block title %}Hello CreerHaieController!{% endblock %}

{% block body %}
    {% block content %}
```

```
<h2>Ajouter un type de haie</h2>
{{ form(form) }}
{% endblock %}
{% endblock %}
```

## Gérer une requête de formulaire

On a affiché un formulaire à partir d'un objet existant. Maintenant on veut récupérer les données venant de l'utilisateur pour créer ou modifier des entités.

- La méthode `handleRequest()` de la classe `Form` permet de récupérer les valeurs des champs dans les inputs du formulaire.
- La méthode `isSubmitted()` de la classe `Form` permet de savoir si on est effectivement en méthode POST.
- La méthode `isValid()` permet de valider les données saisies.

On note que la méthode `handleRequest` est toujours appelée avant la méthode `createView` du formulaire. Ceci pour permettre l'affichage des erreurs de validation dans la vue.

```
$form->handleRequest($request);
if ($form->isSubmitted() && $form->isValid()) {
    // A compléter
}
return $this->render('creer_haie/index.html.twig',
    array('form' => $form->createView())
);
```

# La documentation technique

⚙️ La documentation technique présentera les logiciels permettant la réalisation de notre projet.

## Prérequis

► **Nous allons présenter les prérequis nécessaires à la création de notre sites en différentes sous parties :**

- MAMPP : Serveur Local
- Visual Studio Code
- Notre BDD : PHP MY ADMIN
- Symfony

### MAMPP : Serveur Local

MAMP un ensemble de logiciels MACOS permettant de mettre en place un serveur Web local, un serveur FTP et un serveur de messagerie électronique.



### Visual Studio Code

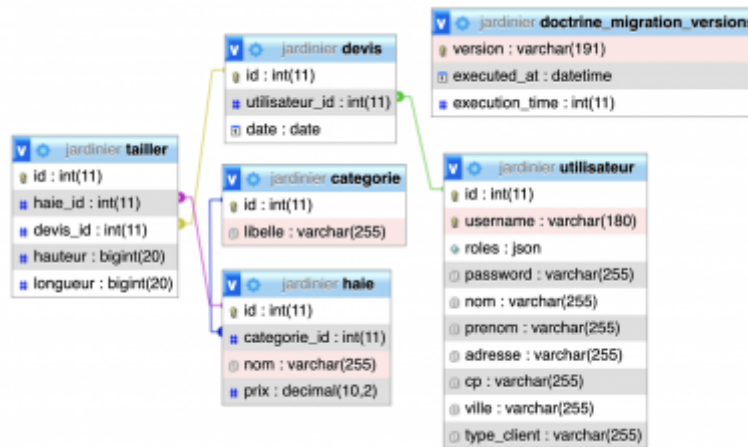
Visual Studio Code est un éditeur de texte générique codé en C++ et Python, disponible sur Windows, Mac et Linux.



### Notre BDD : PHP MY ADMIN

Nous avons également besoin d'une base de données afin de répertorier les haies, utilisateurs, catégories...

Nous avons donc répertorié 6 tables.



Notre base de données est stockée en local grâce à PhpMyAdmin.



## Symfony : Le Modèle-vue-contrôleur

Pour une structure propre et simple, nous avons utilisé le modèle MVC.

L'architecture MVC est l'une des architectures logicielles les plus utilisées pour les applications Web, elle se compose de 3 modules :



- **Modèle** : noyau de l'application qui gère les données, permet de récupérer les informations dans la base de données, de les organiser pour qu'elles puissent ensuite être traitées par le contrôleur.
- **Vue** : composant graphique de l'interface qui permet de présenter les données du modèle à l'utilisateur au sein du code HTML.
- **Contrôleur** : partie principale de l'architecture, car c'est lui qui va faire le lien entre l'utilisateur, le modèle et la vue. Quand l'utilisateur souhaite afficher une page, c'est le contrôleur qui va recevoir cette requête et se charger ensuite de récupérer les données via le modèle, pour les envoyer à la vue qui sera affichée à l'utilisateur. Il est l'intermédiaire entre le modèle et la vue.

# Système d'espace membre

J'ai utilisé la gestion des membres de Symfony :

```
php bin/console make:user
```

Cette commande m'a créé une entité **Utilisateur**, avec un **id**, une username, et **un mot de passe**, et **un tableau de rôle**.

J'y ai donc ajouté les champs nécessaires pour que ce ne soit plus qu'un utilisateur, mais également un Client, c'est à dire les **informations personnelles**, pour cela j'ai utilisé la commande suivante :

```
php bin/console make:entity
```

J'ai implémenté le formulaire d'inscription comme présenté précédemment dans le compte-rendu.

Pour faire le lien entre le contrôleur d'Utilisateur, et mon formulaire de connexion, j'ai configuré le fichier **security.yaml** :

```
main:
    lazy: true
    provider: app_user_provider
    form_login:
        login_path: accueil
        check_path: navbar
        default_target_path: accueil
        always_use_default_target_path: true
    logout:
        path: deconnexion
```

Le **login\_path** est le lieu de redirection après la **connexion** ou **déconnexion**. Le **default\_target\_path** est présent dans le cas où un utilisateur sans permission tente de rentrer sur une page en utilisant le lien. Le **check\_path**, et le name de mon contrôleur, pour accéder au formulaire. Ce formulaire est présent dans un contrôleur nommé **navbar**, qui représente toute ma barre de navigation.

Afin d'ajouter une sécurité supplémentaire, le système permet l'ajout de rôle. J'ai utilisé un rôle qui est **ROLE\_ADMIN**, pour le compte administrateur, seul ce compte a accès à la liste d'utilisateur, à la gestion des haies, et la consultation de tous les devis.

Pour se faire, j'ai été configurer le fichier **security.yaml** et j'y ai ajouté ceci :

```
access_control:
    - { path: ^/utilisateur, roles: ROLE_ADMIN }
```

Dans cet exemple, l'**accès au routage /utilisateur** (et donc tous ses sous-routages comme /utilisateur/1) est limité au **ROLE\_ADMIN**.

Il n'existe donc aucune manière d'accéder à une page sans autorisation.

Comme nous pouvons le voir dans ma table Utilisateur, seul l'administrateur à la **ROLE\_ADMIN** :

				id	username	roles	password	nom	prenom	adresse	cp	ville	type client
<input type="checkbox"/>	Éditer	Copier	Supprimer	1	admin	[]	\$2y\$13\$QZ1ar86ZZcQaSZfV.RamuxGNVXSJwOc3B.pWkzhfk...	dsqdsq	dsqdsq	dsqdsqdsq	33333	dsqdsq	particulier
<input type="checkbox"/>	Éditer	Copier	Supprimer	5	HOW	[]	\$2y\$13\$g61oeWc8A13SiScotKo3KO/wetq5U4aioFh8Qe1dsSv...	NULL	NULL	NULL	NULL	NULL	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	6	dsqdsq	[]	\$2y\$13\$vwthykCPEB6ozsmFI5kiUeG.Vzq[W]YjeVSC0xAVgSF6...	NULL	NULL	NULL	NULL	NULL	NULL

Avec ce système d'authentification, j'ai la possibilité de récupérer l'utilisateur connecté sur n'importe quel contrôleur. Par exemple je l'ai fait sur la page d'accueil de création de devis, pour rappeler l'utilisateur connecté et son type de client.

Pour ce faire j'utilise la fonction suivante :

```
$user = $this->getUser();
```



## Système de gestion des erreurs

Avec **Symfony**, pour détecter des champs vides, nous devons rajouter cet élément ci-dessous :

```
@Assert\NotBlank
```

Nous avons donc rajouté sur tous les champs concernés.

Ensuite, le champ « **code postal** » (interdire d'insérer plus de 5 chiffres, et des lettres) a nécessité quelques modifications. Nous avons modifier la partie qui le concernait dans le fichier **Client.php** dans le dossier **Entity**.

```
/**
 * @ORM\Column(type="decimal", length=5)
 * @Assert\Length(
 *     min = 5,
 *     max = 5,
 *     minMessage = "Le code postal doit comporter {{ limit }}
caractères",
 * )
 */
private $cp;
```

Comme on peut le voir ci-dessus, nous avons modifié le type, sa taille et nous avons rajouté l'élément **@Assert\Length**. Grâce à cela, on ne peut pas ajouter plus de 5 chiffres.

Enfin, afin qu'une erreur apparaisse si nous essayions d'insérer des lettres, nous avons rajouté l'option **invalid\_message** dans le fichier **ClientType** de la class **Form**.

```
->add('prix', MoneyType::class, array('invalid_message'=>'Vous devez
saisir un nombre !'))
```

Pour que ce message d'erreur apparaisse, nous avons rajouté dans le templates **\_form.html.twig** cette petite phrase de code.

```
{{ form_errors(form.cp) }}
```

Nous avons repris cela pour l'ajout et la modification des devis.