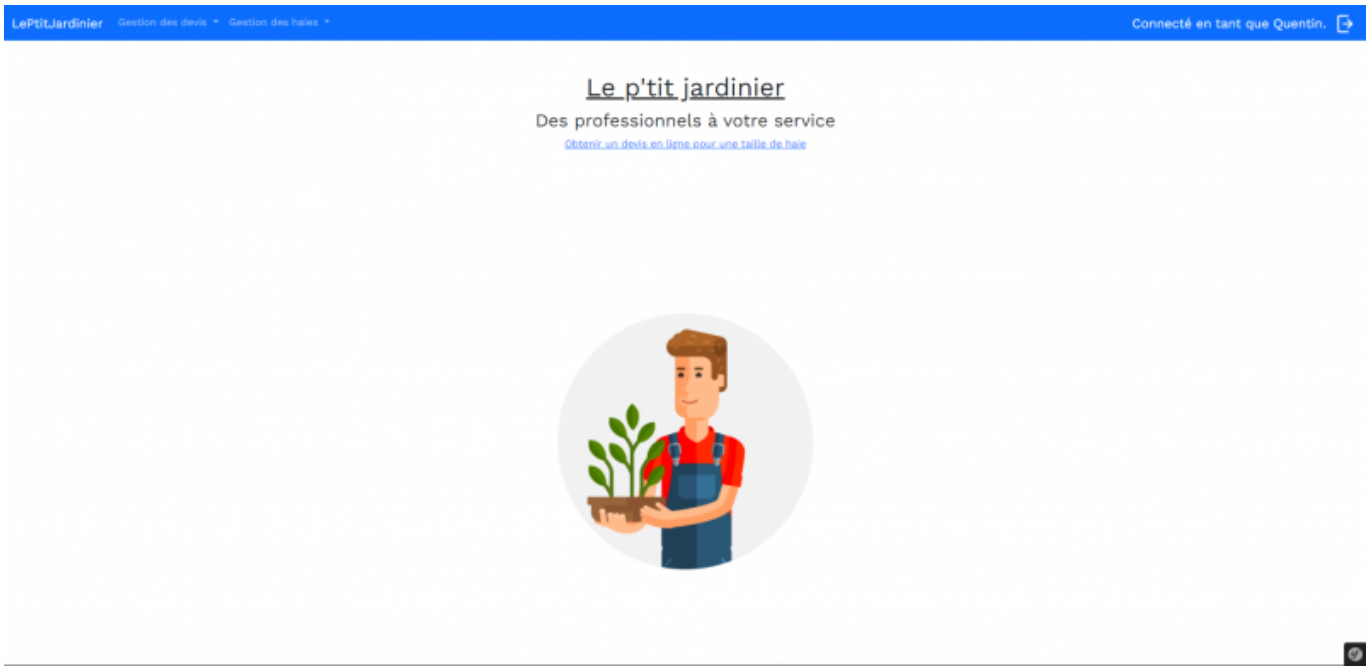
LePtitJardinier - Rodrigues Quentin



Le projet présenté ci-dessous a entièrement été réalisé par moi-même, **Rodrigues Quentin**.

L'objectif de cet **Atelier Professionnel** est la création d'un site de jardinerie, dans le but de créer des devis pour tailler une ou plusieurs **haies**.

Afin de terminer cet atelier, nous avions différentes tâches à réaliser :

- La **gestion des clients**, la création, la modification, la suppression et la modification. Afin de bien affiner ce projet, j'ai choisi en plus de réaliser un **système de connexion** complet.
- La **gestion des devis**, c'est à dire la possibilité pour un utilisateur de créer un devis, avec le choix parmi différentes **haies**, puis la saisie des dimensions, et enfin, la création complète d'une **fiche récapitulatif** avec le détail des prix.
- La **gestion des haies**, ainsi que les **catégories de haie**.

Environnement de travail

Avant de découvrir notre travail, je tiens à vous présenter mon environnement. Nous travaillons à l'aide du base de données MySQL, réalisée en mapping avec doctrine.

Par la suite, nous devions réaliser cette application WEB à l'aide du framework Symfony .

Chaque personne est libre de travail sur son IDE, cependant je préfère travailler sur Visual Studio

Code .

Utilisation du Framework Symfony

Symfony est un ensemble de composants PHP ainsi qu'un **framework MVC libre** écrit en PHP. Il fournit des fonctionnalités **modulables et adaptables** qui permettent de faciliter et d'accélérer le développement d'un site web.

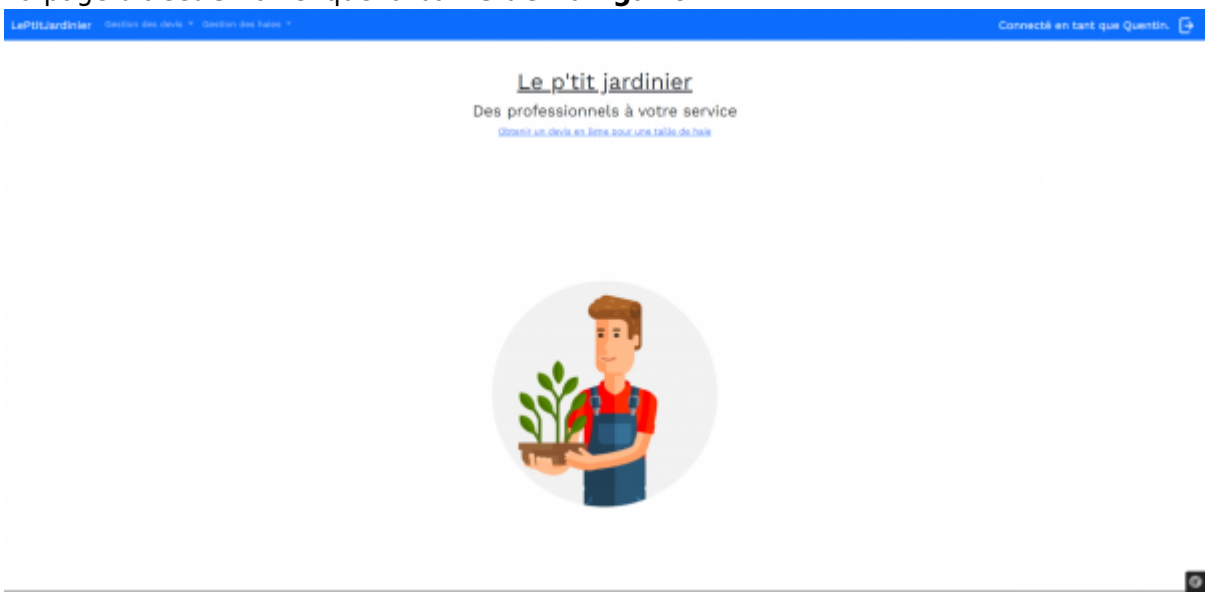
A l'aide de ce framework, j'ai pu facilement réaliser mes différents **routages**, liés à des **controllers**, puis mes **vues** réalisées avec le langage **Twig**.

Il m'a également permis d'utiliser **Doctrine**. Doctrine est un ORM pour PHP. Il s'agit d'un logiciel libre sous licence GNU LGPL. Doctrine est l'ORM par défaut du framework Symfony. Cependant, son utilisation dans le cadre d'un projet développé avec Symfony est optionnelle.

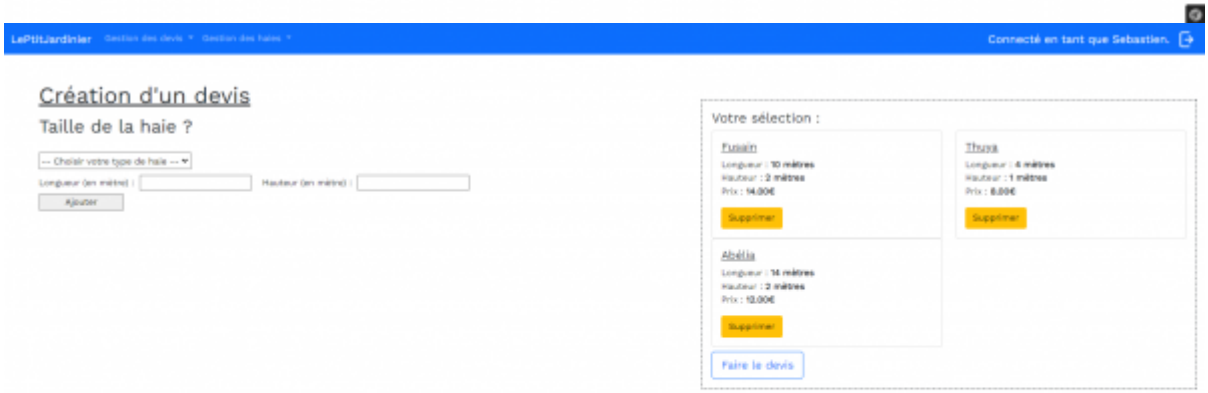
Doctrine m'a permis d'accéder à ma **base de données** sans même devoir faire une requête SQL. Il va automatiquement gérer cela à l'aide d'entité, en programmation **PHP Objet**.

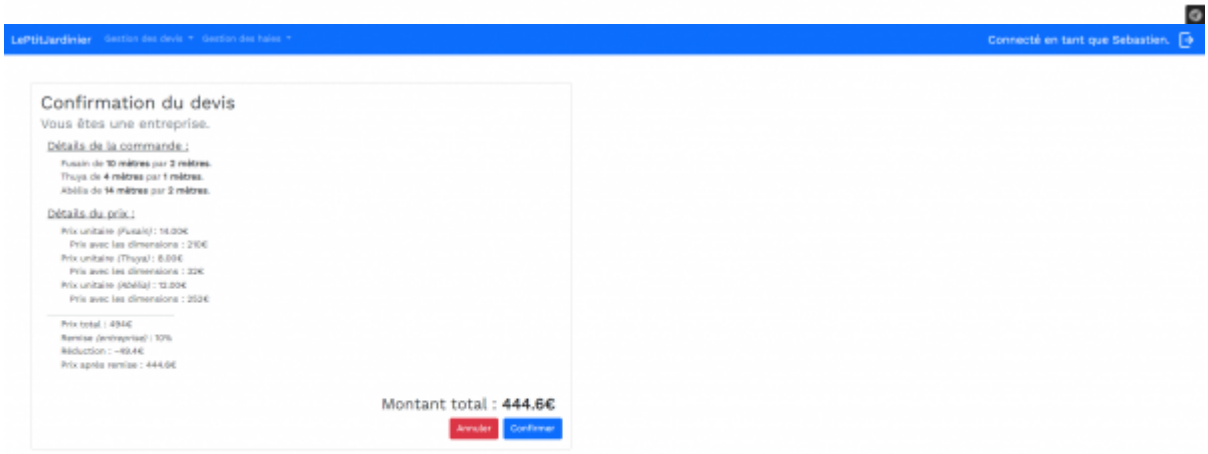
Apparence du site web

La page d'**accueil** ainsi que la **barre de navigation** :

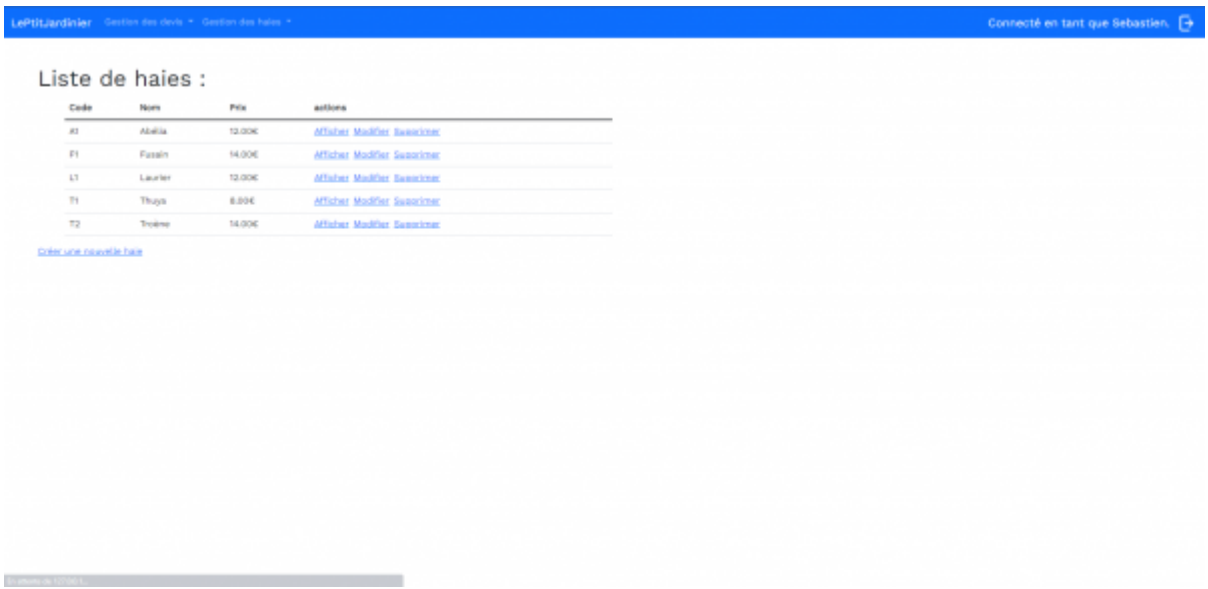


L'onglet **Gestion des devis** :

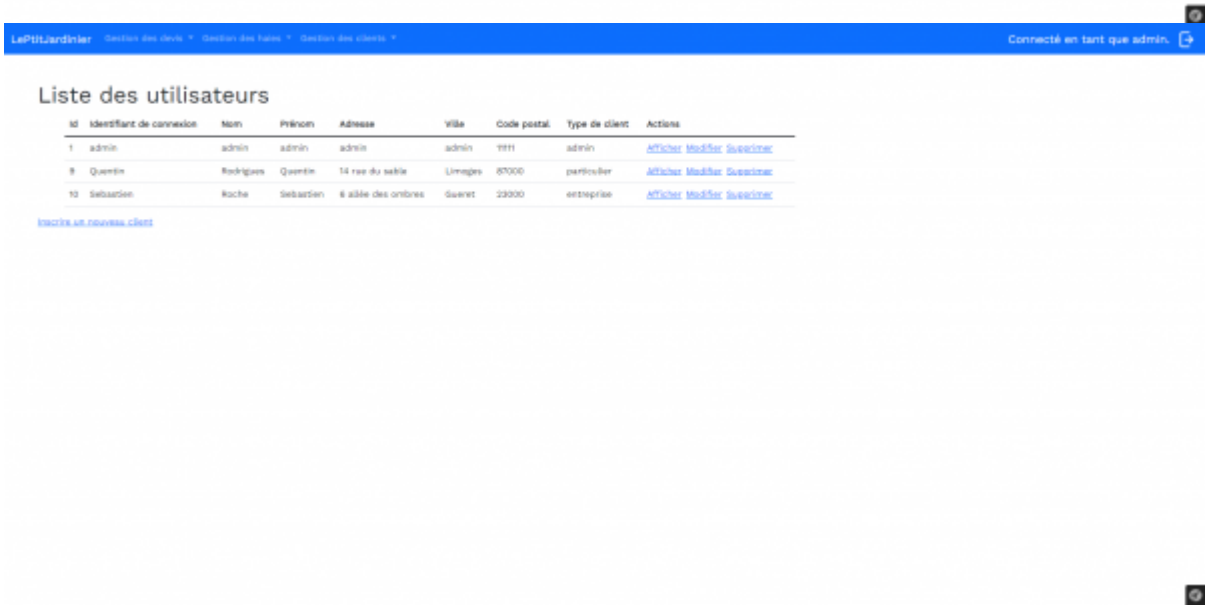




L'onglet **Gestion des haies** :



L'onglet **Gestion des clients** :



Découvrir mon travail

Présentation de la base de données :

- La base de données

Gestion des devis :

- La présentation du type de client
- Le choix des haies et mesures
- La validation du devis

Gestion des haies :

- Le CRUD de la classe HAIE
- Relation avec la catégorie de haie
- Informations sur le formulaire

Gestion des clients :

- Le formulaire d'inscription et le CRUD

Bonus :

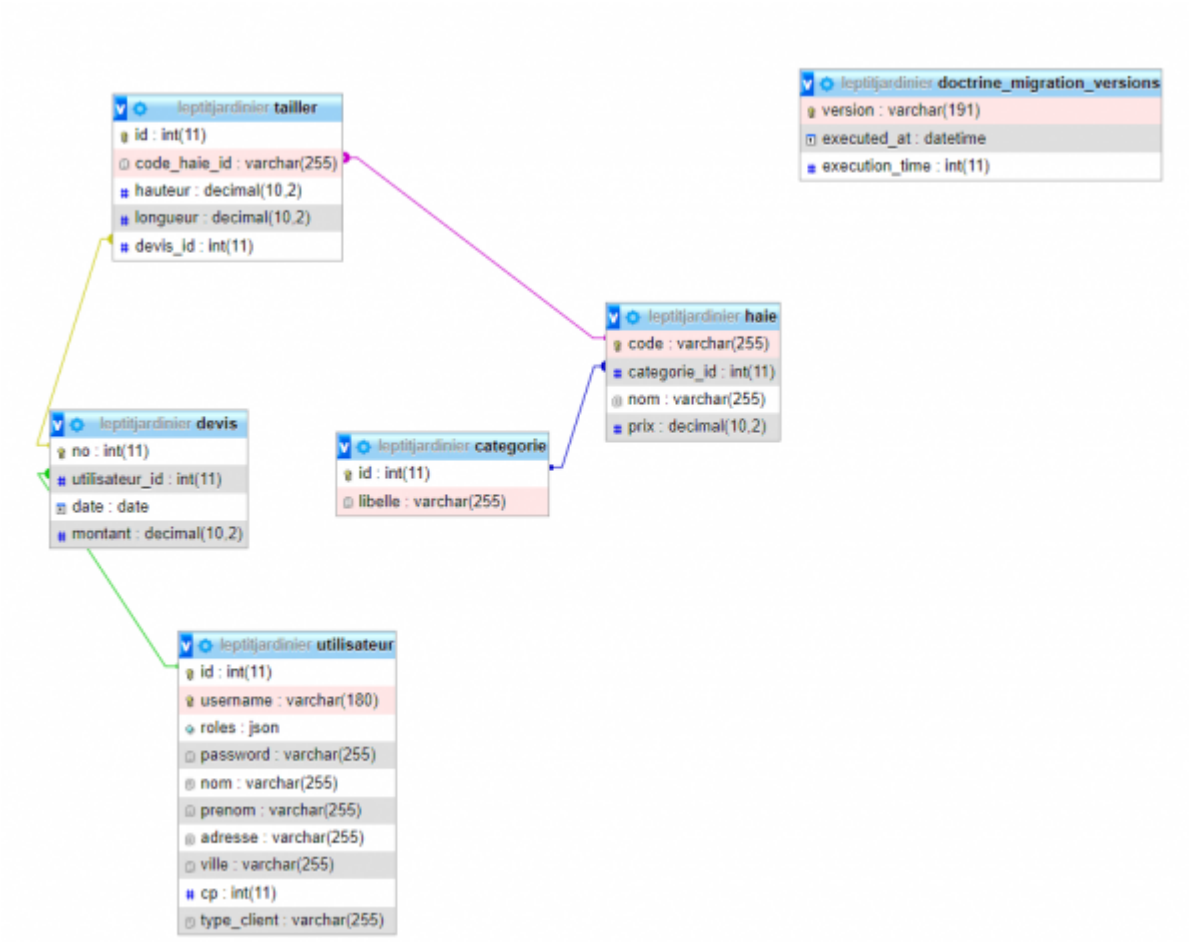
- Le système de connexion et de sécurisation

Conclusion

Pour conclure, cet **Atelier Professionnel** est l'un de mes préférés, en effet nous avons pu découvrir un Framework de grande qualité, et qui sait faire ressortir ses **points forts**.

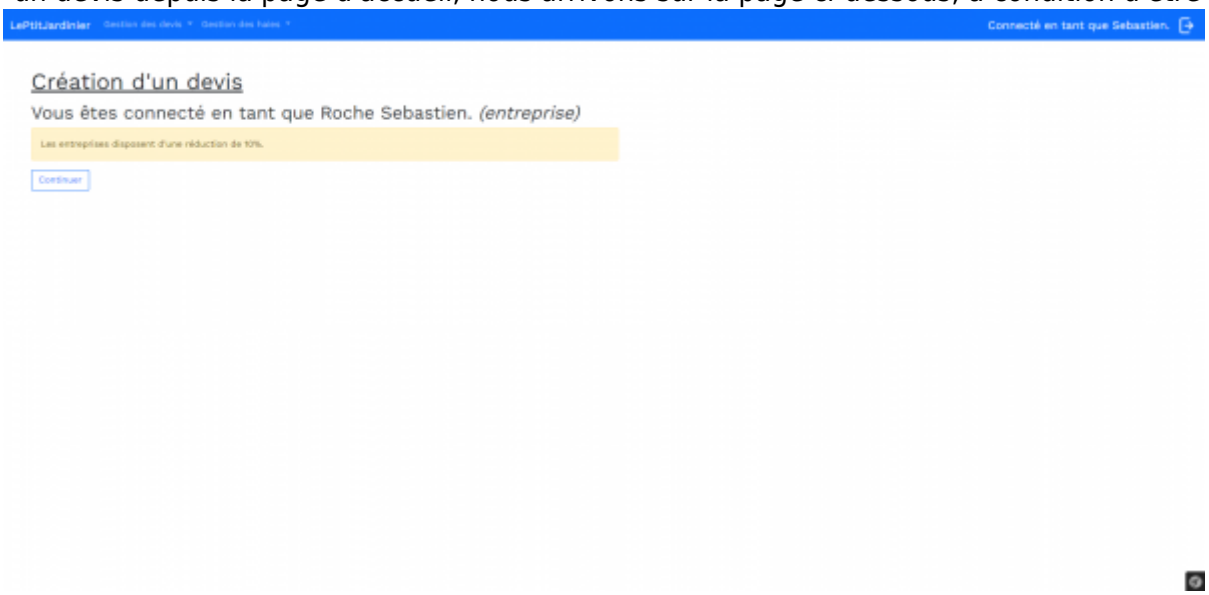
Lors de ma poursuite d'étude, et dans mon futur travail, je serai capable d'utiliser Symfony de manière très **complète**.

Schéma de ma base de données



Création d'un devis

Lorsque l'on clique sur **Nouveau devis** dans la barre de navigation, où que l'on clique directement sur réaliser un devis depuis la page d'accueil, nous arrivons sur la page ci-dessous, à condition d'être connecté :



Cette page rappelle quel utilisateur est connecté, son type, et également si une réduction est disponible ou pas. Voici mon contrôleur :

```
class ChoixController extends AbstractController
{
    #[Route('/choix', name: 'choix')]
    public function index(): Response
    {
        $user = $this->getUser();
        return $this->render('choix/index.html.twig', [
            'controller_name' => 'ChoixController',
            'user' => $user
        ]);
    }
}
```

Ce contrôleur est assez simple à comprendre, je met un routage sur **/choix** pour obtenir l'accès au contrôleur depuis l'URL. Ensuite, je récupère l'utilisateur connecté (Cette partie est disponible plus loin dans le compte-rendu). Puis j'indique au contrôleur ma vue, et j'y transmet le nom de mon contrôleur, et mon utilisateur connecté.

En ce qui concerne **la vue** :

```
{% block content %}
    <style></style>
    <div class="choix-section">
        <h1>Création d'un devis</h1>
        <h2>Vous êtes connecté en tant que
            {{user.nom}}
```

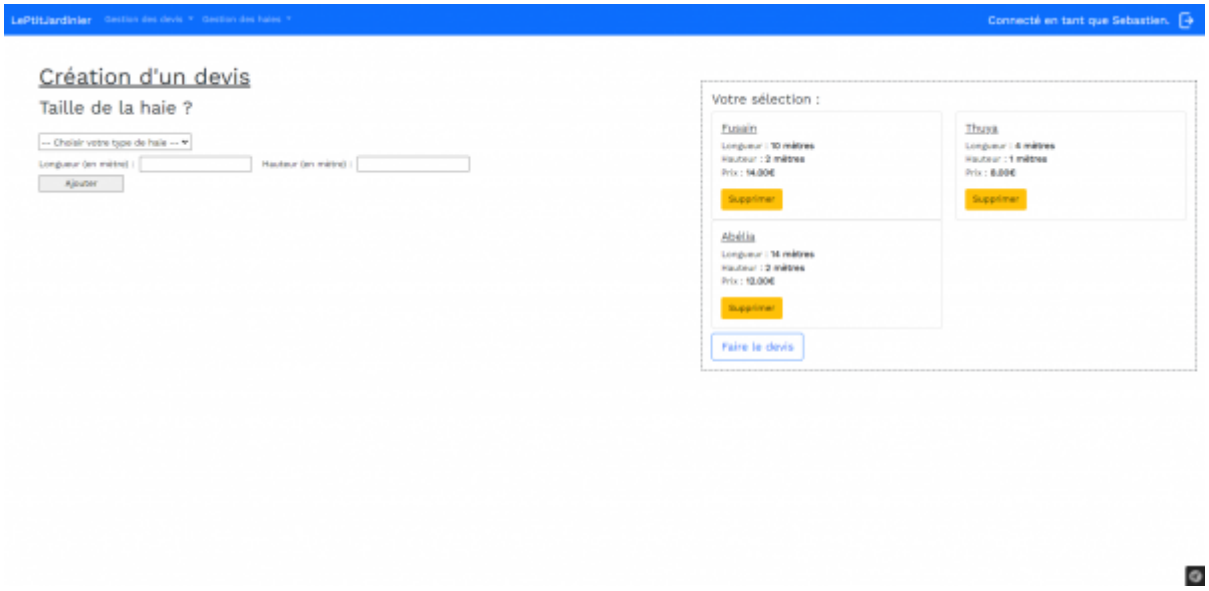
```
            {{user.prenom}}. <i>({{user.typeClient}})</i></h2>
        <div class="alert alert-warning w-50" role="alert">
            {% if user.typeClient == "entreprise" %}
                Les entreprises disposent d'une réduction de 10%.
            {% else %}
                Les particuliers ne disposent pas de réduction.
            {% endif %}
        </div>
        <a class="btn btn-outline-primary" href="{{ path('mesure')
}}">Continuer</a>
    </div>
{% endblock %}
```

Nous pouvons voir que c'est uniquement de l'affichage des variables données par mon contrôleur, car cette page est simplement présente pour faire un rappel des différentes informations.

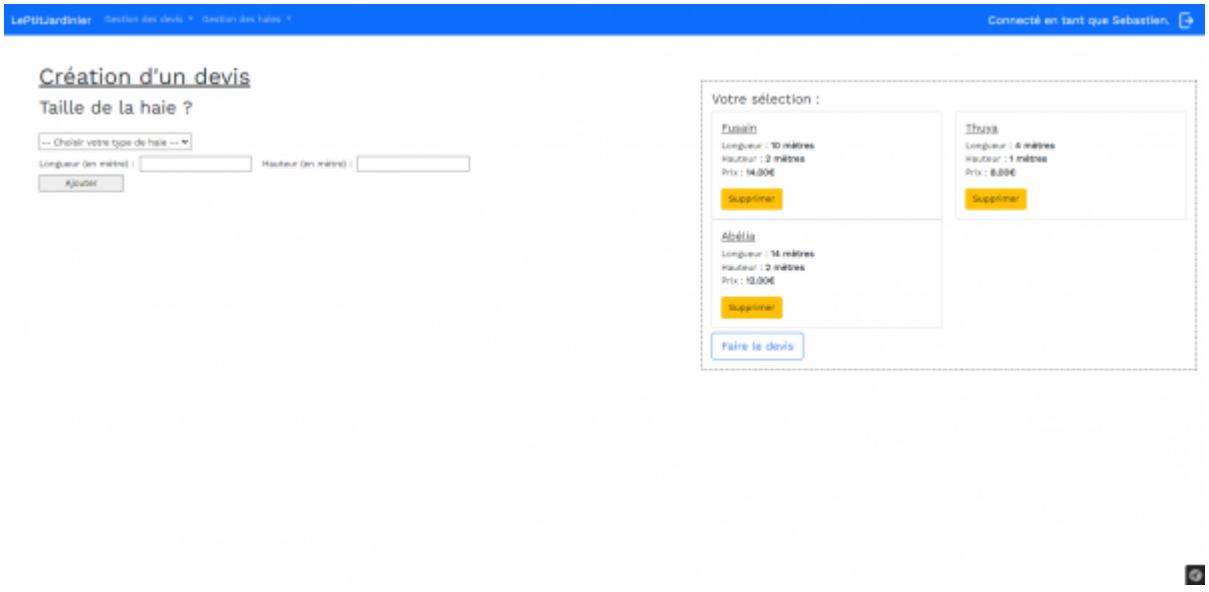
Ensuite, nous arrivons sur la page de sélection et mesure des haies.

Choix et mesures des haies

Après avoir regardé le récapitulatif de quel client est connecté, nous allons pouvoir choisir les haies. Cette partie a été la plus difficile, mais la plus intéressante à mettre en place durant le projet.



Comme nous pouvons le voir, nous arrivons sur un formulaire de choix de haie, et nous devons compléter les haies, ainsi que leurs différentes tailles. Une fois une première haie ajouté, nous obtenons un **panier** contenant la sélection sur la droite :



Nous allons voir comment cela fonctionne, sachant que le tout se passe dans un seul est unique contrôleur.

Le formulaire d'ajout :

```
<div class="d-flex d-inline-flex w-100">
        <div class="container">
            <div class="choix-section">
                <h1>Création d'un devis</h1>
```

```
                <h2>Taille de la haie ?</h2>
                <form class="formMesure" action="{{ path('mesure') }}"
method="post">
                    <select name="typehaie" id="typehaie" required>
                        <option value="" disabled selected>
                            -- Choisir votre type de haie --
                        </option>
                        {% for haie in lesHaies %}
                            <option value="{{ haie.code }}">{{ haie.getNom()
}} ({{haie.prix}}€)</option>
                        {% endfor %}
                    </select><br/>
                    <div class="longueurHauteur">
                        <div class="longueur">
                            Longueur (en mètre) :
                            <input type="number" name="longueur" required/>
                        </div>
                        <div class="hauteur">
                            Hauteur (en mètre) :
                            <input type="number" name="hauteur" required/>
                        </div>
                    </div>
                    <br/>
                    <input type="submit" value="Ajouter"/>
                </form>
            </div>
        </div>
```

Il est assez simple, la partie la plus compliqué va se situer dans le contrôleur :

```
$request = Request::createFromGlobals();

        if ($request->isMethod('POST') &&
is_numeric($request->get('hauteur')) &&
is_numeric($request->get('longueur'))) {

            $typehaie = $request->get('typehaie');
            $lahaie = $this->getDoctrine()
            ->getRepository(Haie::class)
            ->find($typehaie);

            $hauteur = $request->get('hauteur');
            $longueur = $request->get('longueur');
            $laMesure = (object) ['codehaie'=>$lahaie->getCode(),'typehaie'
=> $lahaie->getNom(), 'hauteur' => $hauteur, 'longueur' => $longueur,'prix'
=> $lahaie->getPrix()];
            $mesures[] = $laMesure;
            $session->set('mesures', $mesures);
        }
```

J'interroge le contrôleur pour savoir si une requête **POST** a eu lieu, si c'est le cas, je vérifie également que les deux attribues hauteur et longueur soit bien des nombres, par mesure de sécurité.

Dans le cas où j'ai une requête **POST**, je vais rechercher la haie concerné à l'aide de doctrine et de l'identifiant de la haie (L'identifiant est contenu dans la valeur du sélecteur de haie).

Lorsque j'ai la haie, je vais l'ajouter à un objet nommé **laMesure**, avec des noms de clés spécifique pour y accéder au devis final.

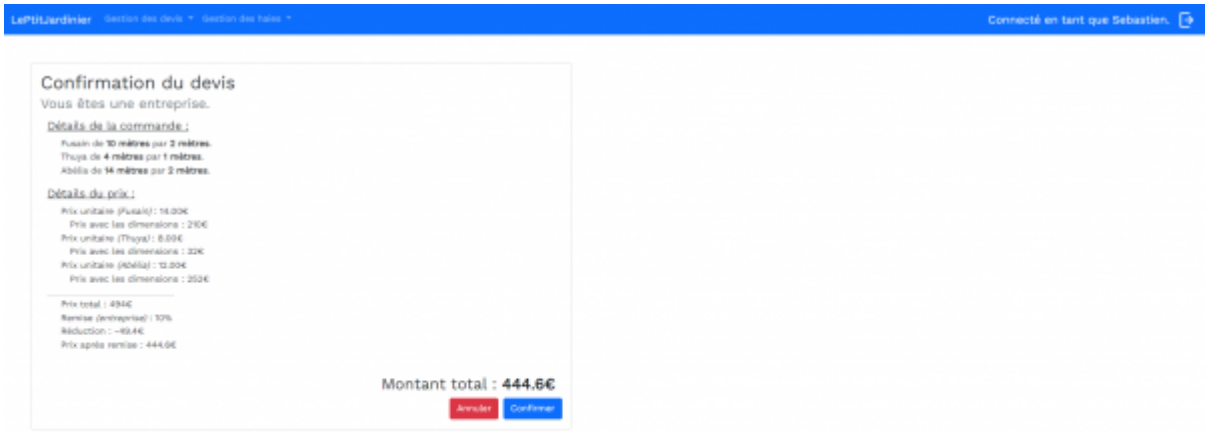
Puis je met à jour mon tableau **mesures**, pour qu'à chaque ajout, le tableau ne se réinitialise pas.

Enfin, je stock le tableau **mesures** dans une **variable session**, pour le garder en mémoire durant le choix de toutes les haies, et surtout pour l'obtenir à la création du devis final.

Voyons maintenant l'affichage du devis final, et l'ajout dans la base de données.

Récapitulatif du devis, et ajout dans la base de données

Après avoir valider le panier, nous obtenons **le devis**, très détaillé, avec une possibilité de l'annuler, ou de le confirmer, et lors de la confirmation, ce devis est inséré dans la base de données.



Je commence par présenter la première partie du contrôleur, c'est ici que le calcul du prix va avoir lieu :

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

        $session = new Session();
        $selection = $session->get('mesures');

        $montantAvantRemise = 0;
        $montantApresRemise = 0;
        $selectionHaies = [];
        $remise = 0;
        $montantRemise = 0;

        foreach ($selection as $mesure) {
            $prixMultiplie = $mesure->prix * $mesure->longueur;
            if($mesure->hauteur > 1.5){
                $prixMultiplie *= 1.5;
            }
            $montantAvantRemise += $prixMultiplie;
            $uneMesure = (object) ['haie' => $mesure->typehaie, 'hauteur' =>
$mesure->hauteur, 'longueur' => $mesure->longueur,
'prixunitaire'=>$mesure->prix, 'prixmultiplie'=>$prixMultiplie];
            $selectionHaies[] = $uneMesure;
        }
```

```
        if($user->getTypeClient() == "entreprise"){
            $remise = 0.10;
        }

        $montantRemise = $montantAvantRemise * $remise;
        $montantApresRemise = $montantAvantRemise - $montantRemise;
```

Le calcul doit respecter certaines conditions :

- Le prix par haie est le prix unitaire de la haie, multiplié par sa longueur.
- Si la haie mesure plus d'1,50m, multiplié le prix par 1,5.
- Si l'utilisateur est une entreprise, appliqué une réduction de 10%.

Dans le contrôleur, je commence par récupérer la **variable session**, dans le but d'obtenir toutes les haies sélectionnées.

Puis j'**initialise** mes variables à 0.

Et je fais le calcul total de toute la sélection, en respectant les **conditions** ci-dessus.

La seconde partie du contrôleur concerne l'annulation ou la validation du formulaire :

```
$request = Request::createFromGlobals();

        if ($request->isMethod('POST')){
            $entityManager = $doctrine->getManager();

            date_default_timezone_set('Europe/Paris');
            $devis = new Devis();
            $devis->setDate(new DateTime());
            $devis->setUtilisateur($user);
            $devis->setMontant($montantApresRemise);

            $entityManager->persist($devis);
            $entityManager->flush();

            foreach ($selection as $uneTaille) {
                $tailler = new Tailler();
                $haie =
$doctrine->getRepository(Haie::class)->find($uneTaille->codehaie);

                $tailler->setCodeHaie($haie);
                $tailler->setHauteur($uneTaille->hauteur);
                $tailler->setLongueur($uneTaille->longueur);
                $tailler->setDevis($devis);

                $entityManager->persist($tailler);
                $entityManager->flush();
            }
```

Nous allons faire comme pour l'ajout du haie dans la panier, c'est à dire vérifier si il y a eu une requête **POST**.

Ensuite, nous pouvons voir la création des différentes class à envoyer dans la base de données, par exemple un **Devis**.

“Tailler” est la table faisant le lien entre un devis, et une ou plusieurs haies. Ainsi, je vais créer un nouvelle **“Tailler”** pour chaque haie sélectionnée. Mais avant cela, je recherche la haie en question à l'aide de doctrine et sa méthode **find()**.

J'utilise la fonction **flush()** de **l'entityManager** pour pousser mes enregistrements dans la base de données.

En ce qui concerne la redirection :

```
         {
            return $this->redirectToRoute('accueil', array('devis' => 1));

        }

        return $this->render('devis/index.html.twig', array(
        'typeuser' => $user->getTypeClient(),
        'remise' => $remise,
        'montantRemise' => $montantRemise,
        'montantApresRemise'=>$montantApresRemise,
        'montantAvantRemise'=>$montantAvantRemise,
        'selectionHaies'=>$selectionHaies
    ));
```

Le premier redirectToRoute permet de revenir à l'accueil en cas de validation, et le paramètre **devis=1** sera passé en GET, pour obtenir une confirmation d'insertion dans l'accueil.

Le second redirectRoRoute est celui par défaut, dans le cas où on vient d'arriver sur la page, sans avoir encore eu la possibilité d'annuler ou confirmer. J'y passe les différents paramètres pour l'affichage.

Et voici une partie de mon twig qui reste assez simple.

```
<div class="row">
                        <div class="col-sm">
                            <h5 class="card-subtitle mb-2 mt-1 text-
decoration-underline">Détails de la commande :</h5>
                            <p class="card-text ms-4">
                                {% for selection in selectionHaies %}
                                {{selection.haie}}
                                de
                                <strong>{{selection.longueur}}
                                    mètres</strong>
                                par
                                <strong>{{selection.hauteur}}
                                    mètres</strong>.<br>
                                {% endfor %}
```

```
                                </p>
                                <h5 class="card-subtitle mb-2 mt-1 text-
decoration-underline">Détails du prix :</h5>
                                <p class="card-text ms-4">
                                    {% for selection in selectionHaies %}
                                        Prix unitaire
<i>({{selection.haie}})</i> : {{selection.prixunitaire}}€<br>
                                            <span class="ms-3">Prix avec les
dimensions : {{selection.prixmultiplie}}€</span>
                                        <br>
                                    {% endfor %}
                                        <hr class="my-1 w-25">
                                        <div class="ms-4">
                                        Prix total : {{montantAvantRemise}}€<br>
                                        {% if not remise == 0 %}
                                        Remise <i>({{typeuser}})</i> : {{remise
* 100}}%<br>
                                        Réduction : -{{montantRemise}}€<br>
                                        Prix après remise :
{{montantApresRemise}}€
                                        {% endif %}
                                        </div>
                                </p>
                            </div>
```

Et voilà, la partie d'insertion de devis est entièrement fonctionnelle.

CRUD de la class HAIE

Une des tâches demandées était la gestion des haies. Pour cela, j'ai choisi d'utiliser la fonction **CRUD** de Symfony :

```
php bin/console make:crud
```

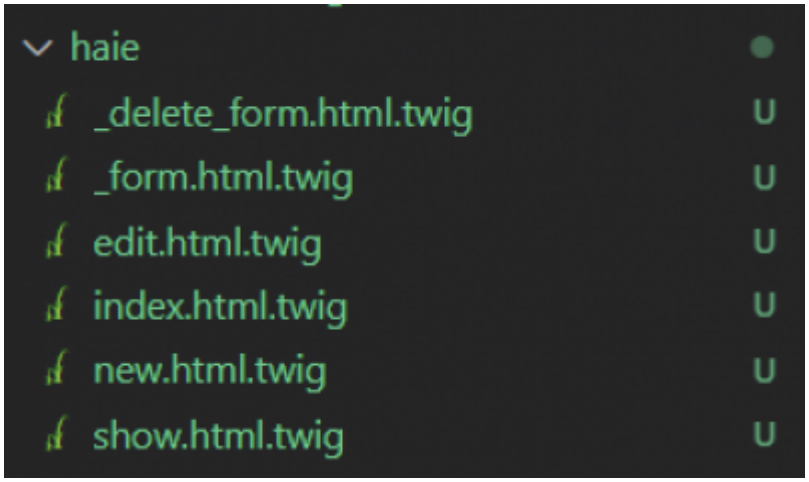
Cette commande est très **puissante** car elle permet de créer automatiquement le contrôleur et les vues pour la création, modification, suppression, et l'affichage des haies. De plus elle créé le formulaire type d'une haie.

Voici le contrôleur créé à l'aide de cette commande :

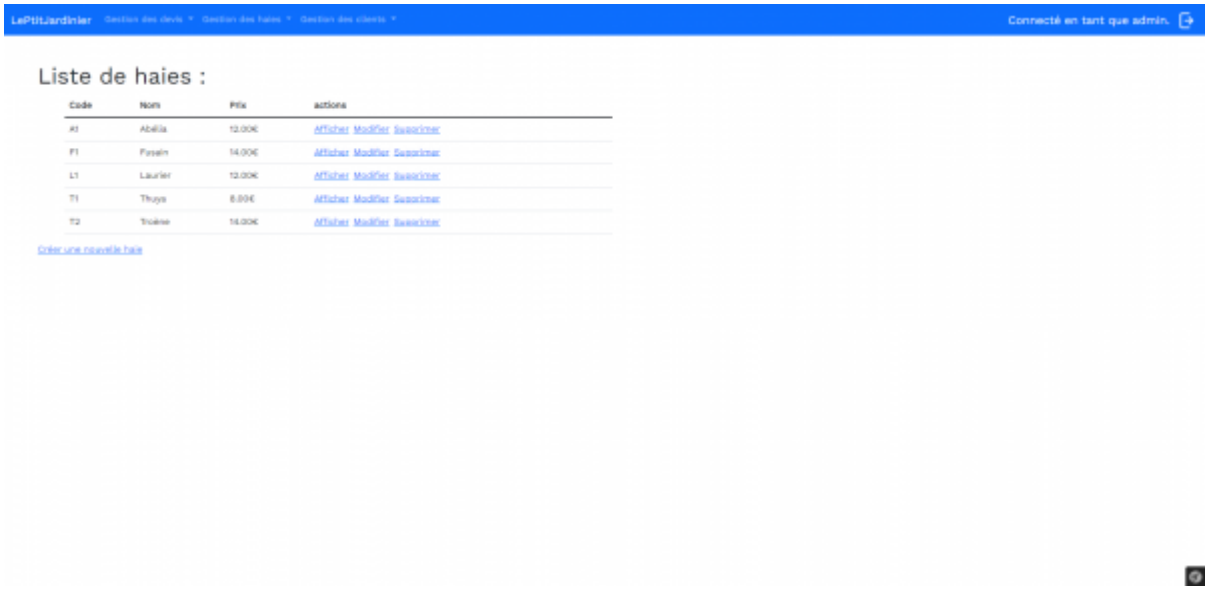
```
#[Route('/', name: 'haie_index', methods: ['GET'])]
    public function index(HaieRepository $haieRepository): Response
    {
        return $this->render('haie/index.html.twig', [
            'haies' => $haieRepository->findAll(),
        ]);
    }

    #[Route('/new', name: 'haie_new', methods: ['GET', 'POST'])]
    public function new(Request $request, EntityManagerInterface
$entityManager): Response
    {
        $haie = new Haie();
        $form = $this->createForm(Haie1Type::class, $haie);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $entityManager->persist($haie);
            $entityManager->flush();

            return $this->redirectToRoute('haie_index', [],
Response::HTTP_SEE_OTHER);
        }

        return $this->renderForm('haie/new.html.twig', [
            'haie' => $haie,
            'form' => $form,
        ]);
    }

    #[Route('/{code}', name: 'haie_show', methods: ['GET'])]
    public function show(Haie $haie): Response
    {
        return $this->render('haie/show.html.twig', [
            'haie' => $haie,
        ]);
```

```
    }

    #[Route('/{code}/edit', name: 'haie_edit', methods: ['GET', 'POST'])]
    public function edit(Request $request, Haie $haie,
EntityManagerInterface $entityManager): Response
    {
        $form = $this->createForm(Haie1Type::class, $haie);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $entityManager->flush();

            return $this->redirectToRoute('haie_index', [],
Response::HTTP_SEE_OTHER);
        }

        return $this->renderForm('haie/edit.html.twig', [
            'haie' => $haie,
            'form' => $form,
        ]);
    }

    #[Route('/{code}', name: 'haie_delete', methods: ['POST'])]
    public function delete(Request $request, Haie $haie,
EntityManagerInterface $entityManager): Response
    {
        if ($this->isCsrfTokenValid('delete'.$haie->getCode(),
$request->request->get('_token'))) {
            $entityManager->remove($haie);
            $entityManager->flush();
        }

        return $this->redirectToRoute('haie_index', [],
Response::HTTP_SEE_OTHER);
    }
```

Nous pouvons voir que tous les routages sont parfaitement réalisés. Et également, les vues qui sont très bien organisées :



Et voici un exemple d'affichage de la liste, avec l'ajout d'un peu de **css** :



Après avoir montrer la réalisation du CRUD pour les haies, vous pouvez découvrir la relation avec la catégorie d'une haie.

Le lien entre la haie, et sa catégorie

Une haie est représentée par une classe avec doctrine. Ainsi, pour créer une **clé étrangère** sur celle-ci il va falloir être assez **rigoureux**.

La commande suivante permet de générer une entité, je l'ai utilisé pour gérer la relation entre une haie et une catégorie de haie, sachant qu'une haie à obligatoirement une catégorie, et une catégorie peut appartenir à aucune ou plusieurs haies :

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

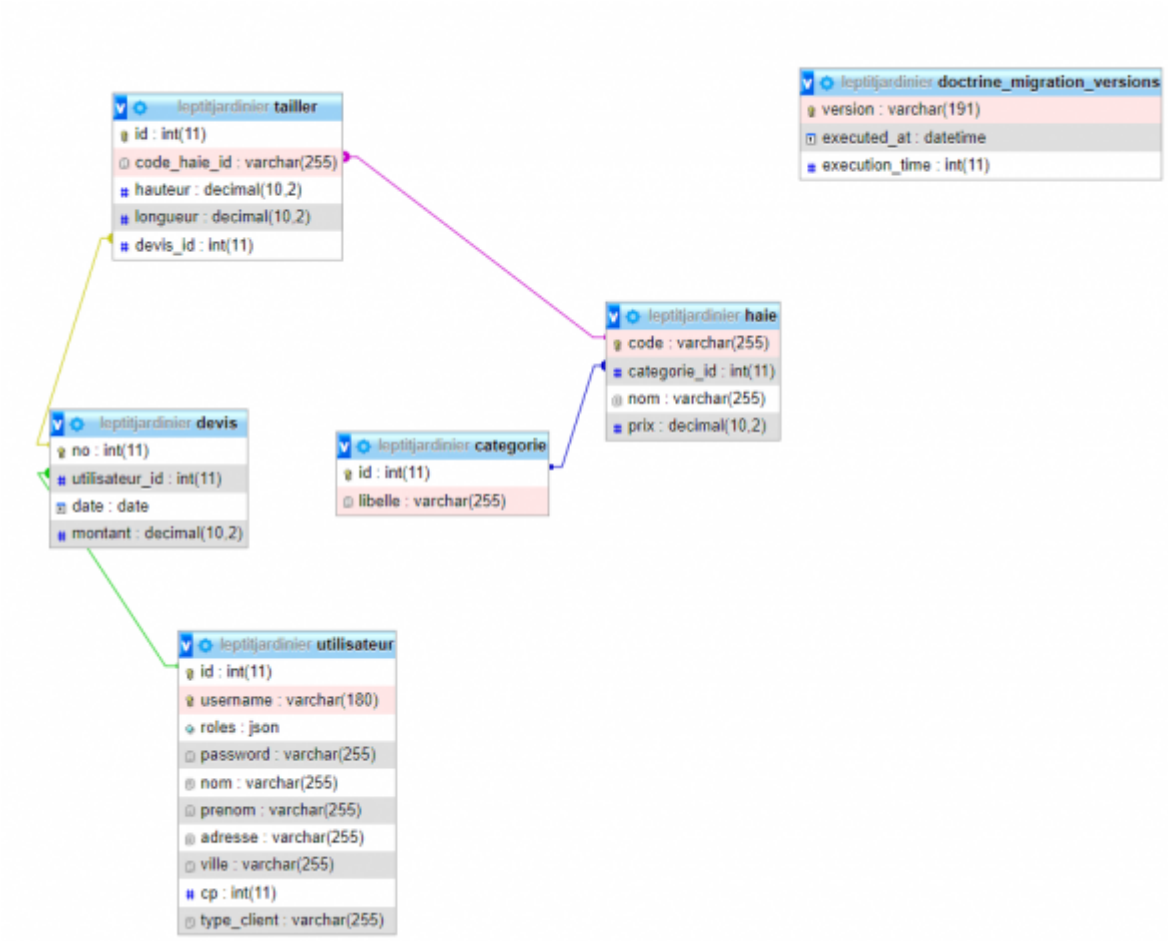
Doctrine va créer un objet Haie, et voici la méthode qu'il utilisé pour faire le lien entre deux entités.

```
/**
     * @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;
    }
```

La relation entre ces deux tables se fait parfaitement lorsqu'on migre la base de données MySql à l'aide de ces deux commandes :

```
php bin/console doctrine:migrations:diff
php bin/console doctrine:migrations:migrate
```

Et voici le rendu sur le concepteur de PhpMyAdmin :



Et voilà, la relation fonctionne parfaitement, et Symfony gère les cas d'erreurs dans le cas où certains champs sont manquant.

Pour continuer la découverte du projet, vous pouvez voir plus d'informations sur le formulaire de Haie.

Le formulaire type d'une haie

Le formulaire type d'une haie est généré en même temps que le CRUD avec la commande suivante :

```
php bin/console make:crud
```

Analysons ce formulaire :

```
public function buildForm(FormBuilderInterface $builder, array $options):
void
    {
        $builder
            ->add('code', TextType::class, array('label' => 'Code haie : ',
'attr' => array('class'=>'input-group w-25', 'placeholder'=>'Code')))
            ->add('nom', TextType::class, array('label' => 'Nom : ', 'attr'
=> array('class'=>'input-group w-25', 'placeholder'=>'Nom')))
            ->add('prix', NumberType::class, array('label' => 'Prix : ',
'attr' => array('class'=>'input-group w-25', 'placeholder'=>'Prix')))
            ->add('categorie')
        ;
    }
```

Il est assez simple, il **build** un formulaire et ajoute chaque input avec **→add()**. Pour ma part, j'y ai ajouté un attribut **class** pour le stylisé.

Ce formulaire est appelé en Twig de cette manière :

```
{{ form_start(form) }}
```

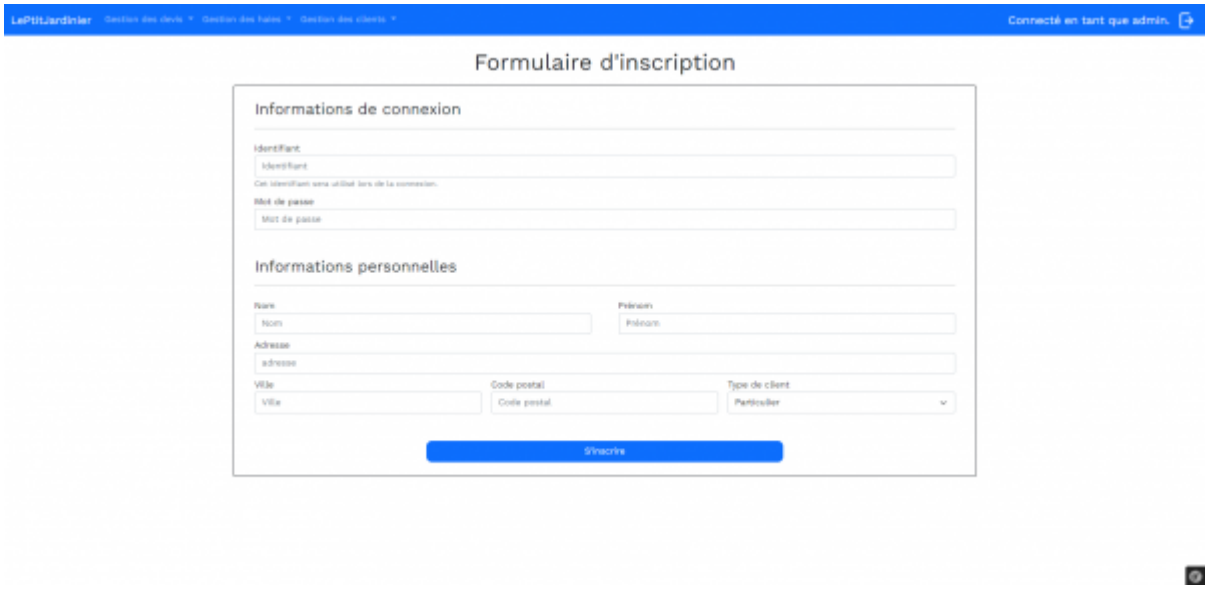
Le formulaire est fonctionnel. Découvrir le formulaire d'inscription.

Le formulaire d'inscription

Afin de m'occuper de la **gestion des clients**, j'ai décidé de voir les choses plus grand. En effet, j'ai implanté un système de connexion complet, et le système d'inscription permet à la fois de s'inscrire en tant que **client**, et en tant qu'**utilisateur**.

Le système de connexion est présenté en bonus dans ce compte-rendu, ici, j'évoque uniquement la gestion des clients

Le formulaire est comme ceci :

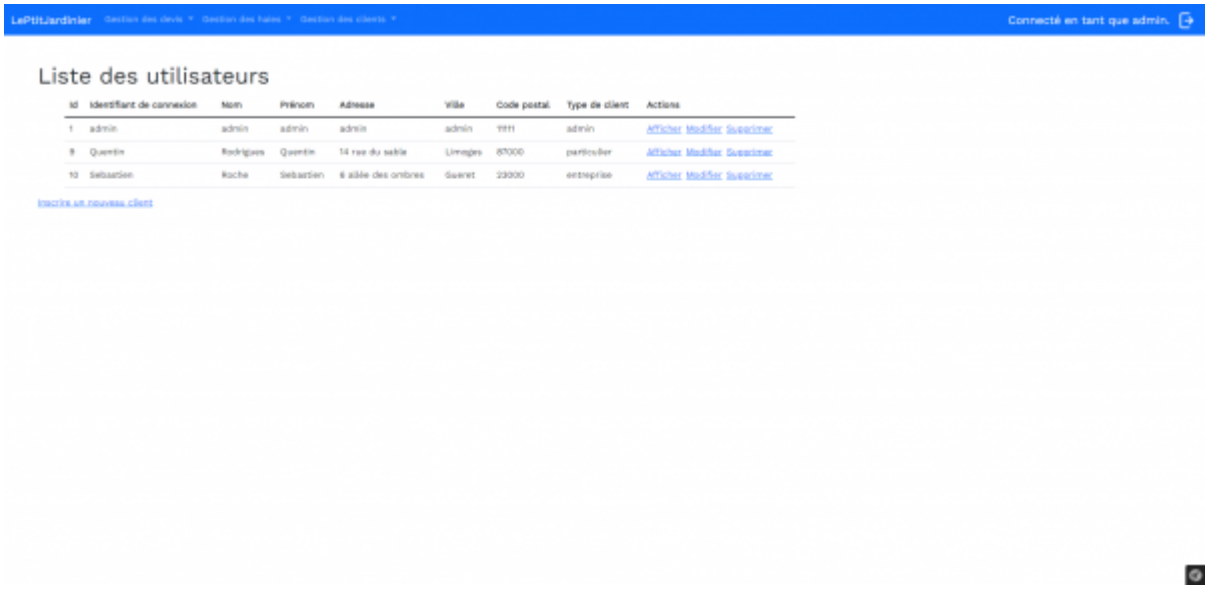


Pour réaliser ce formulaire, je n'ai pas utilisé la commande de CRUD, mais la commande spéciale de gestion d'utilisateur de Symfony :

```
php bin/console make:registration-form
```

Cette commande m'a généré un formulaire, qui prend en compte la sécurité et donc crypte également le mot de passe.

En ce qui concerne le reste de la gestion des clients, j'ai utilisé la méthode CRUD évoqué précédemment.



Poursuivre sur plus de détails sur le système de connexion.

L'implémentation d'un système de connexion sécurisé

Nous avions comme mission la gestion des clients, et j'ai choisi d'en faire un système de connexion en même temps, pour que chaque client ait la possibilité de créer leurs devis, sans passer par une sélection de client.

Anthony Rodrigues, étudiant en BTS SIO, m'a aidé à implémenter le **système d'authentification sécurisé** créer par Symfony. Ainsi, j'ai utilisé la commande suivante pour initialiser ce système :

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

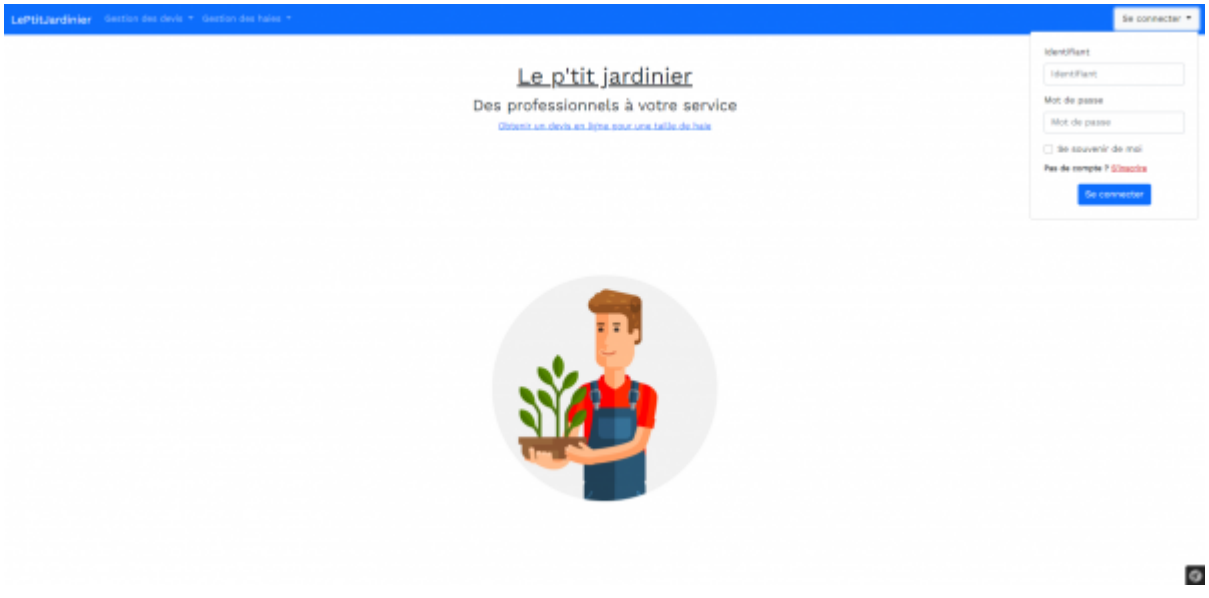
Cette commande m'a créé une entité **Utilisateur**, avec un **id**, une username, et **un mot de passe**, et **un tableau de rôle**.

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.

J'ai également ajouté le formulaire de connexion. Il se trouve directement dans la barre de navigation pour qu'il soit accessible partout.



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	ville	cp	type_client
Éditer Copier Supprimer	1	admin	["ROLE_ADMIN"]	$2y$13$l8BuD7zDynR3KDgJWXMZOs5ajlzKLHdM.swroyq7bD...	admin	admin	admin	admin	11111	admin
Éditer Copier Supprimer	9	Quentin	[]	$2y$13$lcPOo5C1PrV8k97ccwUSMuDbqZUv0XIDXsk6MsjL37i...	Rodrigues	Quentin	14 rue du sable	Limoges	87000	particulier
Éditer Copier Supprimer	10	Sebastien	[]	$2y$13$e9oS0Lem4ZSHr4M8p8hwruIFW7WjpC7fYazJwf9Rlj.d...	Roche	Sebastien	6 allée des ombres	Gueret	23000	entreprise

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();
```