DOCUMENTATION TECHNIQUE

Le P'tit Jardinier | Projet avec Symfony

Le P'tit Jardinier

Application de création de devis

Flavien DEROY
BTS SIO SLAM
Lycée Suzanne Valadon

TABLE DES MATIERES

CONTEXTE ... 2
CAHIER DES CHARGES ... 2
CAPTURES D'ECRAN PRINCIPALES ... 4
1. CREER UN PROJET SUR SYMFONY ... 5
1.1. Créer un projet ... 5
1.2. Créer des vues/contrôleurs/entités ... 6
1.2.1. Vue/Contrôleur ... 6
1.2.2. Entités (base de données) et migration ... 6
2. MODELE CONCEPTUEL DES DONNEES ... 8
3. CONNEXION / INSCRIPTION DES UTILISATEURS ... 12
3.1. L'utilisateur ... 12
3.2. La connexion ... 13
3.3. L'inscription ... 15
4. SECURITE ET ACCESSIBILITE ... 18
4.1. Permissions d'accès ... 18
4.2. Accès aux pages ... 19
4.3. Mots de passe hachés ... 19
CREATION D'UN DEVIS ... 20
GESTION D'UN DEVIS ... 23
GESTION DES UTILISATEURS ET DES HAIES (ADMINISTRATEURS) ... 25
TESTS ET VERIFICATION ... 27
CONCLUSION ... 27

CONTEXTE

La société « Le P'tit Jardinier » a pour activité l'entretien des jardins pour les particuliers et les organisations. Etant contactée fréquemment pour des devis de taille de haies, elle souhaite mettre en place une application Web permettant à toute personne d'obtenir un tarif et de créer/supprimer un devis. Cette dernière doit aussi permettre aux administrateurs de l'application de pouvoir directement gérer les utilisateurs, devis et haies.

CAHIER DES CHARGES

Objectif du projet

L'intérêt de ce projet est de permettre la connexion et inscription d'utilisateurs qui pourront obtenir un tarif selon la taille et type des haies et ainsi générer un devis selon ces caractéristiques. Cependant, le but est aussi d'ajouter, pour les administrateurs, des outils de gestion pour les haies et les utilisateurs inscrits et même, détenir tous les devis enregistrés sur l'application.

Mais qu'est-ce que Symfony et comment il fonctionne ?

Symfony est un **Framework** web écrit en modèle de conception **MVC** (Modèle-Vue-Contrôleur) utilisant le langage **PHP** qui facilite le développement d'applications web performantes et maintenables. Il repose sur des **composants réutilisables (librairies / packages / bundles)** et suit les bonnes pratiques du développement web. Avec Symfony, vous pouvez créer des projets complexes en utilisant des conventions plutôt que de configurer chaque détail. Symfony intègre également **Composer**, un gestionnaire de dépendances, ce qui simplifie l'intégration de bibliothèques tierces. Grâce à sa popularité, Symfony dispose d'une grande **communauté de développeurs** et d'un **écosystème** riche en **bundles et composants** pour répondre aux besoins spécifiques des projets. En résumé, Symfony offre une approche structurée et rapide pour créer des sites internet dynamiques.

Outils et langages utilisés

Dans le cadre de la conception de ce projet, j'ai utilisé mon logiciel de développement habituel Visual Studio Code, où l'installation et liaison de ces extensions/logiciels sont primordiale pour que Symfony fonctionne correctement, car ils fournissent les bases nécessaires au développement et à l'exécution des applications Symfony :

- Installer PHP 8.0.13 ou une version ultérieure. La plupart des installations PHP récentes ont ces extensions activées par défaut, mais il est important de vérifier que votre version de PHP correspond aux exigences de Symfony.

- Installer Composer : Composer est un gestionnaire de dépendances utilisé par Symfony pour installer les packages PHP nécessaires à votre application. Composer est un logiciel libre écrit en PHP. Vous pouvez l'installer en téléchargeant le fichier exécutable approprié pour votre système d'exploitation à partir du site officiel de Composer. Une fois installé, Composer

mettra à jour le PATH système pour que vous puissiez exécuter la commande "composer" à partir de n'importe quel répertoire de votre ligne de commande.

- Installer Symfony CLI : Symfony CLI est un outil qui fournit un ensemble complet d'outils pour développer et exécuter des applications Symfony localement. Vous pouvez l'installer en téléchargeant le binaire "symfony" correspondant à votre système d'exploitation à partir du site officiel de Symfony. Une fois installé, vous pourrez utiliser la commande "symfony" pour exécuter diverses tâches de développement et de gestion de votre application Symfony.

Les objectifs / besoins du projet pour l'utilisateur :

Calcul de devis et possibilité de générer un devis : Récupérer les informations insérées par le client dans le formulaire de statut du client et mesures de la haie pour effectuer un calcul qui sera le montant du devis et pouvoir générer un devis avec ces mêmes informations.

Consultation des devis : L'utilisateur pourra consulter la liste des devis qu'il a créé dans l'application.

Les objectifs / besoins du projet pour l'administrateur :

Consultation des devis : Ici, la différence entre les utilisateurs lambdas et les administrateurs se fait sur la consultation des devis. Les clients (utilisateurs lambdas) détiendront une liste comportant uniquement leurs devis alors que les administrateurs détiendront tous les devis de tous les clients dans la liste.
Création et gestion des haies : la gestion (ajout, création, modification, suppression et affichage de la liste des haies) s'est faite avec l'outil de création automatique de CRUD dont dispose Symfony. Les administrateurs seront les seuls à détenir ces fonctionnalités.
Consultation de la liste des clients : Toujours grâce au système de CRUD dont dispose Symfony, une liste des clients est créée et seuls les administrateurs pourront voir cette liste.

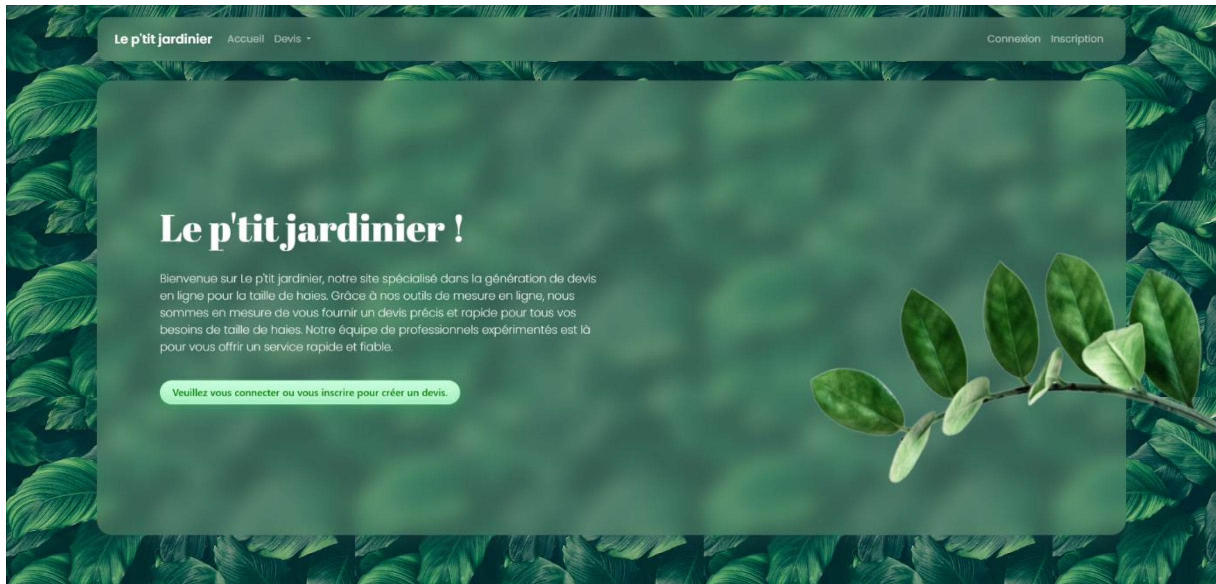
Ce que gère Symfony dans ce projet :

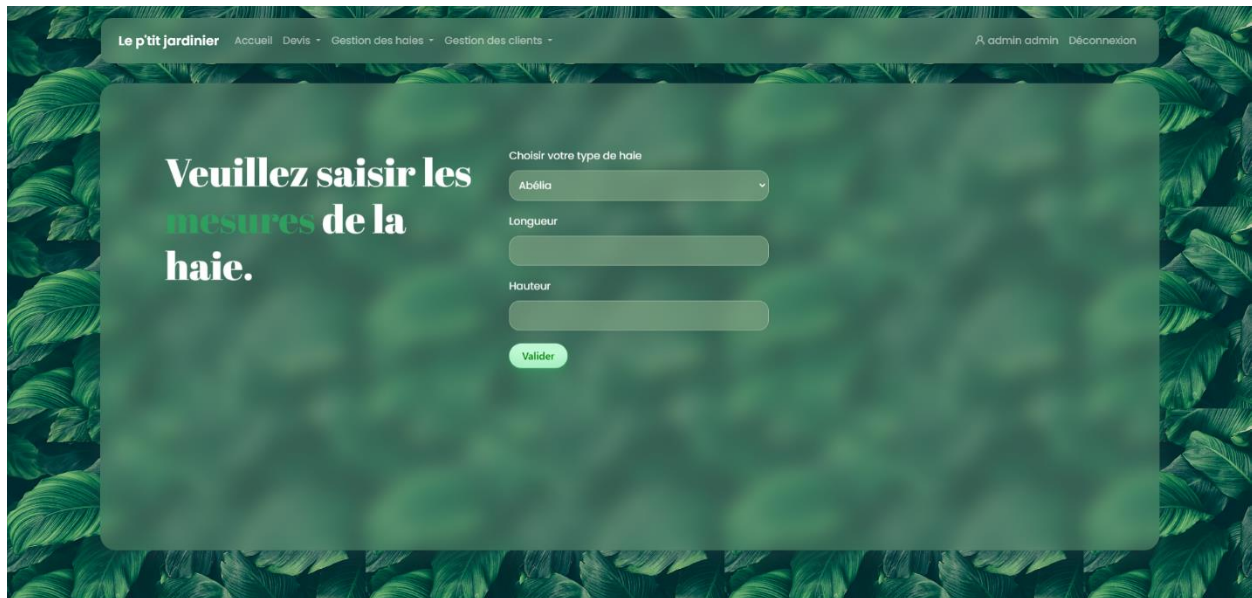
Gestion des rôles des utilisateurs du site : Deux types d'utilisateurs, les « users » qui n'ont seulement accès à la création et la gestion de leur(s) devis, ainsi que les « administrateurs » qui ont eux aussi accès à la création mais à la gestion des devis de tous les clients inscrits en bdd. Ils ont aussi accès d'ajouter et à la gestion de la liste des haies disponibles ainsi que la consultation de la liste des clients.

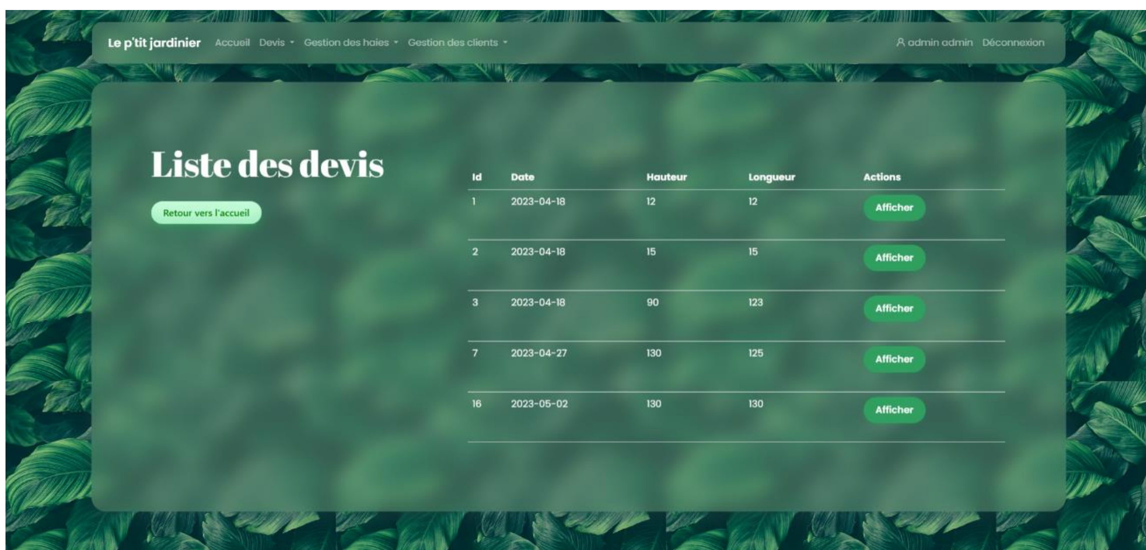
Mots de passe : Les mots de passe sont hachés et salés dans la bdd automatiquement grâce à Symfony.

Base de donnée : Symfony offre la fonctionnalité de création automatique des bases de données en fonction des configurations définies dans les fichiers « Entity », ce qui facilite la mise en place et la gestion des bases de données.

CAPTURES D'ECRAN PRINCIPALES







1. CREER UN PROJET SUR SYMFONY

1.1. Créer un projet

Une fois tous les prérequis installés sur votre machine locale, il y a plus qu'à créer un projet Symfony dans le Terminal de Visual Studio Code. Cependant, vérifiez si votre ordinateur détient toutes les exigences à l'aide de cette commande :

```
symfony check:requirements
```

Si tout est bien installé, la création du projet peut commencer avec cette commande qui va totalement créer un squelette d'application Symfony :

```
symfony new monProjet --full
```

NOTE :

Le squelette va créer toute une liste de bundles et le projet Symfony sera initialisé automatiquement par Composer dans un dossier (monProjet dans cette exemple).

Le squelette ressemble à cela :

```
> bin
> config
> migrations
> public
> src
> templates
> tests
> translations
> var
> vendor
  .env
  .env.test
  .gitignore
  composer.json
  composer.lock
  docker-compose.override.yaml
  docker-compose.yaml
  phpunit.xml.dist
  symfony.lock
```

Vous pouvez maintenant lancer votre application Symfony en utilisant le serveur web intégré en tapant la commande suivante :

```
symfony server:start
```

Par la suite, ouvrez un navigateur web et accédez à l'adresse indiquée dans le terminal pour voir votre application Symfony en cours d'exécution (souvent : 127.0.0.1:8000/accueil)

1.2. Créer des vues/contrôleurs/entités

1.2.1. Vue/Contrôleur

La partie Vue/Contrôleur permet de gérer les fonctionnalités/utilisations de l'application web.

La vue se créée automatiquement dans le dossier templates selon le nom du Controller. Nous créons un Controller à l'aide de cette commande :

```
symfony console make:controller
```

Un Controller gère les requêtes HTTP et coordonne les actions nécessaires pour traiter ces requêtes et renvoyer une réponse. En outre, c'est celui qui gère l'envoie des données vers la vue.

1.2.2. Entités (base de données) et migration

Créer une Entity va permettre de créer tout simplement une table dans la base de données. On gère les données directement dans les fichiers afin de coordonner comme il faut et éviter des conflits. Pour créer une entité, on utilise la commande :

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

Ce procédé marche comme l'ajout d'une table dans le SGBD, mais là, dans un terminal.

Il créé par nom, type, taille et autres des colonnes.

```
Class name of the entity to create or update (e.g. AgreeableGnome):
> MonEntite

created: src/Entity/MonEntite.php
created: src/Repository/MonEntiteRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> MonChamp1

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/MonEntite.php

Add another property? Enter the property name (or press <return> to stop adding fields):
>
```

Cela dit, pour pouvoir envoyer les entités dans une base de données, il faut que dans le fichier .env, le lien envoyant au bon SGBD soit bien exacte.

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"
```

Dans mon cas, je souhaite l'envoyer dans mon SGBD phpMyAdmin dans la bdd « jardinier » donc le lien devra ressembler à cela :

```
DATABASE_URL="mysql://root@127.0.0.1:3306/jardinier?serverVersion=5.7"
```

Et grâce à cela, la migration des entités pourra donc se faire.

Lorsque l'ensemble des classes entités sont créées et bien relié, il suffit de migrer à l'aide de ces deux commandes :

```
php bin/console make:migration

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

2. MODELE CONCEPTUEL DES DONNEES

Dans le cas de ce projet, j'avais besoin de quatre tables : Devis, User (Utilisateur), Haie, Catégorie.

Il fallait que :

- ➔ Devis soit relié à la table User (user_id en référence à id de la table User) pour relié un ou plusieurs devis à un utilisateur
- ➔ Devis soit relié à la table Haie (haie_id en référence à id de la table Haie) pour relié le type de haie correspondante pour le devis (important pour le prix du devis)
- ➔ Haie soit relié à la table Catégorie (categorie_id en référence à id de la table Categorie) car une haie appartient à une catégorie de haie.

Pour se faire, il suffit simplement de créer à l'aide de la commande exprimés auparavant, `php bin/console make:entity` pour les quatre table. Cependant, comment marche les liaisons de clés étrangères ?

Cela marche dans 2 méthodes différentes :

- Soit par le terminal, en attribuant dans « Property name » le nom de la table à laquelle s'associer et dans « Field type » mettre le champs « relation » (au lieu de string, integer, ...) et d'attribuer le type de relation (ManyToOne, OneToMany, ManyToMany, OneToOne).

- Soit par code où l'on attribut le type, la relation en appelant la classe comme ici :

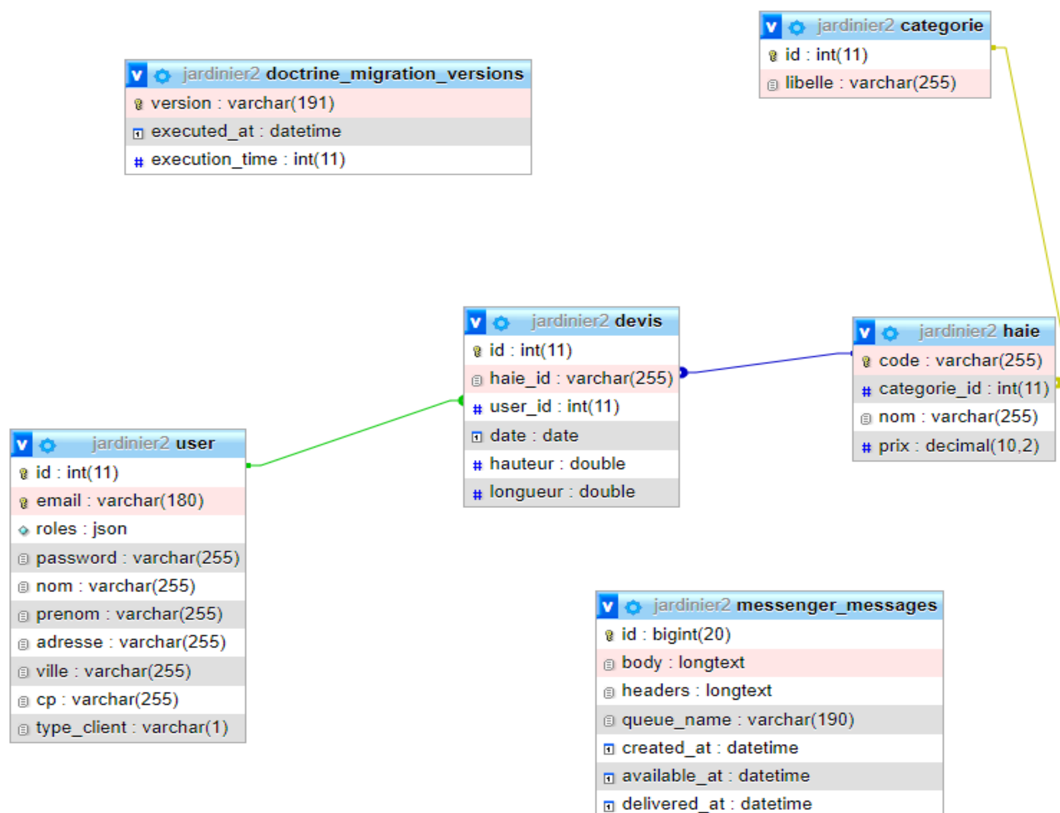
```
#[ORM\ManyToOne(inversedBy: 'devis')]
    #[ORM\JoinColumn(nullable: false)]
    private ?User $user = null;
```

et en appelant ses getters et setters correspondants :

```
public function getUser(): ?User
    {
        return $this->user;
    }

    public function setUser(?User $user): self
    {
        $this->user = $user;

        return $this;
    }
```

Une fois que tous ces éléments sont bien paramétré, vous n'avez plus qu'à migrer les données à l'aide des commandes cités plus tôt.

Ce qui a donc donné ce résultat :



Les tables doctrine_migration_versions et messenger_messages sont des tables automatiques de Symfony qui n'ont pas grand interêt.

La table « doctrine_migration_versions » permet de garder une trace des migrations de base de données appliquées à l'application Symfony, garantissant ainsi une gestion cohérente et contrôlée du schéma de base de données.

La table « messenger_messages » est utilisée par le composant Messenger de Symfony, qui offre une fonctionnalité de messagerie asynchrone. Ce composant permet d'envoyer des messages dans des files d'attente et de les traiter de manière asynchrone par des « workers » (des exécuteurs de tâches). La table « messenger_messages » est utilisée pour stocker les messages en attente de traitement.

Avant de commencer à expliquer comment j'ai conceptualiser mon application, voici un exemple d'une classe entité :

```php
#[ORM\Entity(repositoryClass: DevisRepository::class)]
class Devis
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(type: Types::DATE_MUTABLE)]
    private ?\DateTimeInterface $date = null;

    #[ORM\Column]
    private ?float $hauteur = null;

    #[ORM\Column]
    private ?float $longueur = null;

    #[ORM\ManyToOne(inversedBy: 'devis')]
    #[ORM\JoinColumn(nullable: false,referencedColumnName:"code")]
    private ?Haie $haie = null;

    #[ORM\ManyToOne(inversedBy: 'devis')]
    #[ORM\JoinColumn(nullable: false)]
    private ?User $user = null;

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getDate(): ?\DateTimeInterface
    {
        return $this->date;
    }

    public function setDate(\DateTimeInterface $date): self
    {
        $this->date = $date;

        return $this;
    }

    public function getHauteur(): ?float
    {
        return $this->hauteur;
    }
```

```php
    public function setHauteur(float $hauteur): self
    {
        $this->hauteur = $hauteur;

        return $this;
    }

    public function getLongueur(): ?float
    {
        return $this->longueur;
    }

    public function setLongueur(float $longueur): self
    {
        $this->longueur = $longueur;

        return $this;
    }

    public function getHaie(): ?Haie
    {
        return $this->haie;
    }

    public function setHaie(?Haie $haie): self
    {
        $this->haie = $haie;

        return $this;
    }

    public function getUser(): ?User
    {
        return $this->user;
    }

    public function setUser(?User $user): self
    {
        $this->user = $user;

        return $this;
    }
}
```

3. CONNEXION / INSCRIPTION DES UTILISATEURS

Cette partie est essentielle car le but est qu'un utilisateur puisse générer son devis depuis un tarif calculé. Le point fort de Symfony est d'avoir de nombreux composants puissants, comme celui de créer directement une page de connexion et d'inscription à partir d'une table.

3.1. L'utilisateur

Cela commence par la création d'un utilisateur (et non une table utilisateur !) où on lui admet des identifiants de connexion :

```
$ symfony console make:user
```

```
The name of the security user class (e.g. User) [User]:

 > User



 Do you want to store user data in the database (via Doctrine)? (yes/no)
[yes]:

 > yes



 Enter a property name that will be the unique "display" name for the user
(e.g. email, username, uuid) [email]:

 > email



 Will this app need to hash/check user passwords? Choose No if passwords are
not needed or will be checked/hashed by some other system (e.g. a single sign-
on server).



 Does this app need to hash/check user passwords? (yes/no) [yes]:

 > yes
```

```
 created: src/Entity/User.php

 created: src/Repository/UserRepository.php

 updated: src/Entity/User.php

 updated: config/packages/security.yaml
```

La commande ajoute également la configuration d'un fournisseur d'utilisateurs dans votre configuration de sécurité (security.yaml).

```
# config/packages/security.yaml

security:

    # ...

providers:

        # used to reload user from session & other features (e.g. switch_user)

        app_user_provider:

            entity:

                class: App\Entity\User

                property: email
```

3.2. La connexion

La création de page de connexion se fait à l'aide d'une commande :

```
$ symfony console make:auth
```

Elle permet tout simplement de :

- mettre à jour la configuration de sécurité,
- générer un template pour la connexion
- créer une classe d'authentification (authenticator)

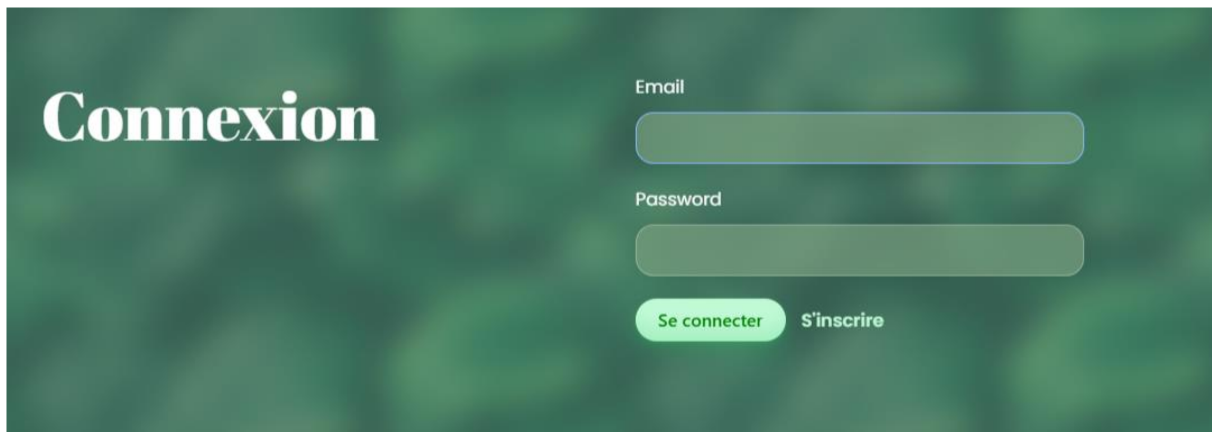
Cela est d'un gain de temps énorme.

Cette commande va lancer un assistant qui va vous demander de renseigner les informations suivantes :

- Le type d'authentification (avec ou sans formulaire de connexion)
- Le nom de la classe contenant l'authentification (UserAuthenticator dans mon exemple suivant)
- Le nom du contrôleur qui contiendra les routes de connexion et déconnexion (SecurityController)
- La création ou non d'une route de déconnexion (/logout)

```
-   What style of authentication do you want? [Empty authenticator]:

-     [0] Empty authenticator

-     [1] Login form authenticator

-    > 1

-

-    The class name of the authenticator to create (e.g.
    AppCustomAuthenticator):

-    > User

-

-    Choose a name for the controller class (e.g. SecurityController)
    [SecurityController]:

-    >

-

-    Do you want to generate a '/logout' URL? (yes/no) [yes]:

-    > yes

-

-    created: src/Security/UserAuthenticator.php

-    updated: config/packages/security.yaml

-    created: src/Controller/SecurityController.php

-    created: templates/security/login.html.twig
```

C'est ce qui a permis de générer cette page simplement (outre le CSS que j'ai fait moi-même).



3.3. L'inscription

Pour l'inscription, même principe, instaurez cette commande :

```
$ symfony console make:registration-form
```

Cette commande va lancer un assistant qui va vous demander de renseigner les informations suivantes :

- Veut-on ajouter une annotation (@UniqueEntity) dans notre classe User pour les rendre uniques
- Veut-on envoyer un email aux utilisateurs pour activer leur compte
- Veut-on connecter automatiquement les utilisateurs après leur inscription

```
-   Creating a registration form for App\Entity\User

-

-    Do you want to add a @UniqueEntity validation annotation on your User
    class to make sure duplicate accounts aren't created? (yes/no) [yes]:

-    > yes

-

-    Do you want to send an email to verify the user's email address after
    registration? (yes/no) [yes]:

-    > no

-
```

```
-    Do you want to automatically authenticate the user after registration?
    (yes/no) [yes]:

-    > yes

-

-    updated: src/Entity/User.php

-    created: src/Form/RegistrationFormType.php

-    created: src/Controller/RegistrationController.php

-    created: templates/registration/register.html.twig
```

C'est ce qui aussi permet l'élaboration presque automatique de cette page entière :

Cependant, nous souhaitons savoir si le type du compte utilisateur est une « Entreprise » ou un « Particulier ». J'ai donc ajouté une autre vue avec son controller qui permettra de mettre en session le choix du statut choisie pour l'implémenter dans le formulaire d'inscription et qui donc gardera la donnée et l'implémentera comme toute les autres en bdd.

Une variable de session est une variable stockée sur le serveur web pour stocker des données temporaires pendant une session utilisateur.

Dans le cas de choix du statut, cela a été géré comme ceci :

```
use Symfony\Component\HttpFoundation\Session\Session;
```

Ne pas oublier d'utiliser le use correspondant

Dans ChoixController, je récupère donc le choix grâce aux getters et je l'implémente dans ma session :

```
        $request = Request::createFromGlobals();
         $choix=$request->get('choix');
         $session = new Session();
         $session->set('choix', $choix);
```

Et dans le Form de l'inscription, j'ajoute un champs caché qui permet d'ajouter les données lors de l'ajout en bdd :

```
            ->add('typeClient', TextType::class, [
                'attr' => ['class' => 'invisible'],
                'label_attr' => ['class' => 'invisible'],
                'label' => '',
                'data' => $choix,
            ])
```

C'est la classe « invisible » qui va permettre de cacher l'input et son label afin de pas générer de problème/conflit de compréhension pour l'utilisateur, en display : none ; tout en gardant l'utilité d'appeler la variable $choix.

D'ailleurs, la variable $choix récupère les valeurs des boutons radios (p et e) :

```
                    action="{{ path('app_register') }}" method="post">
                    {# Le formulaire est obligatoire pour la redirection vers 
l'autre page lors de la validation de formulaire (les zones de textes, ...) #}
                    <div class="card">
                        <input type="radio" id="particulier" name="choix" 
value="p">
                        <label for="particulier">
                            <div class="card-text">
                                <h3>Particulier</h3>
                                <p>Vous n'obtiendrez pas de remise.</p>
                            </div>
                        </label>
                    </div>

                    <div class="card">
                        <input type="radio" id="entreprise" name="choix" 
value="e">
                        <label for="entreprise">
                            <div class="card-text">
                                <h3>Entreprise</h3>
                                <p>Vous obtiendrez une remise de 10%.</p>
```

```
                        </div>
                    </label>
                </div>
```

Qui servira pour le typeClient :

Éditer Copier Supprimer 3 deroy.flavien@gmail.com [] $2y$13$kNNmizgEl4hNcy3vench/.b65eYx4PHtUOiNCH977tU... Deroy Flavien 5 Rue de Rochechouart Limoges 8700 e

4. SECURITE ET ACCESSIBILITE

4.1. Permissions d’accès

Avant de parler de l’élaboration de chaque fonctionnalité de l’application, je vais expliquer comment j’ai séparé les fonctionnalités entre celles des administrateurs et des utilisateurs lambdas.

Afin de gérer ces permissions d’accès à certaines pages de l’application, j’ai utilisé le système de rôle de base de Symfony. En effet lorsqu’on créer l’entité User, on retrouve un champ « rôle » qui permet de gérer dans le code les permissions. Les crochets vides [] définissent le rôle utilisateur (ROLE_USER au niveau du code). Si on veut que le compte de la table User soit administrateur, alors on change directement dans le SGBD les crochets en « ROLE_ADMIN »:

roles
[]
["ROLE_ADMIN"]
[]
[]

Ainsi je peux gérer les permissions de mes pages dans les contrôleurs et dans les vues par exemple à l’aide de la méthode is_granted() qui permet de savoir si l’utilisateur en train de parcourir l’application possède un certain rôle :

```
   {% if is_granted('ROLE_ADMIN') %}
       <a class="dropdown-item"
href="{{path('app_devis_crud_index')}}">Consulter tous les devis</a>
   {% elseif is_granted('ROLE_USER')%}
       <a class="dropdown-item"
href="{{path('app_liste_devis_pour_un_utilisateur', {'id':
app.user.id})}}">Consulter mes devis</a>
   {% else %}
       <a class="dropdown-item" href="{{path('app_login')}}">Consulter mes
devis</a>
   {% endif %}
```

J’ai utilisé ces techniques afin de gérer les permissions de toute mon application.

4.2. Accès aux pages

L’accès aux pages sur Symfony se fait à l’aide des routes :

```
class AcceuilController extends AbstractController
{
    #[Route('/accueil', name: 'app_accueil')]
    public function index(): Response
    {
        return $this->render('acceuil/index.html.twig', [
            'controller_name' => 'acceuilController',
        ]);
    }
}
```

4.3. Mots de passe hachés

Symfony propose un composant appelé "Security" qui inclut un mécanisme de hachage des mots de passe, utilisant généralement l'algorithme bcrypt, pour garantir la sécurité des mots de passe dans les applications web développées avec ce Framework.

On l’active à l’aide de cette commande dans le Terminal :

```
composer require symfony/password-hasher
```

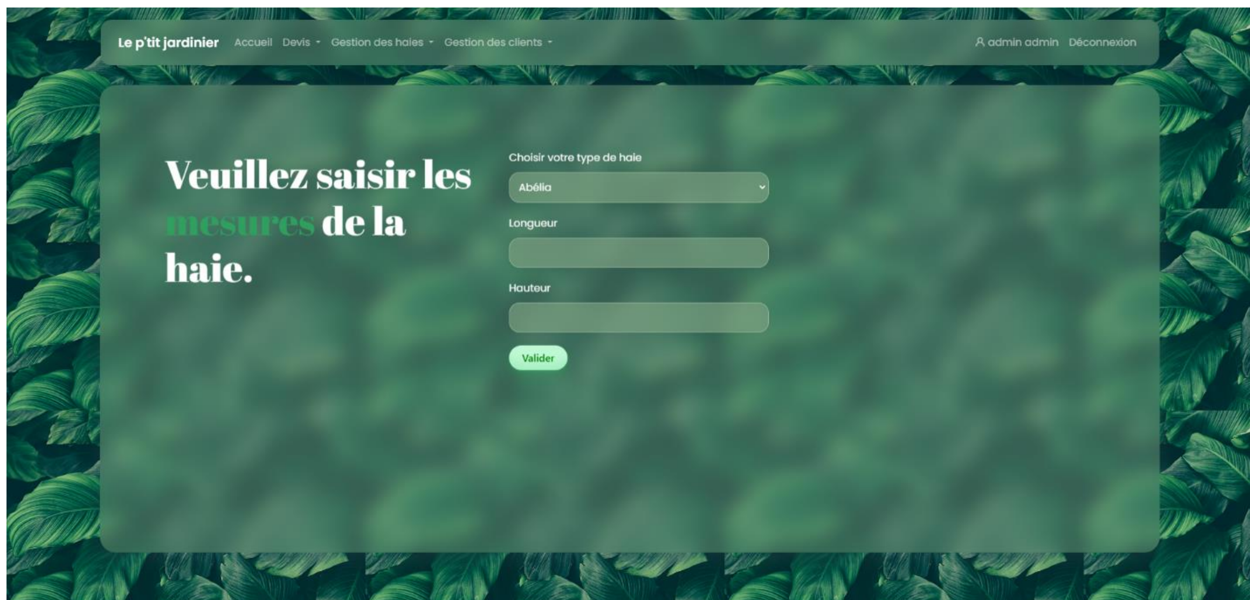
On peut apercevoir lorsqu’on inscrit des utilisateurs, que le hachage se fait correctement en bdd :

password	nom	prenom	a
$2y$13$UMPxTugqgst1Id0bw25uFOybtHMMi4IGLAg9k/eZXdF...	Jules	Artaud	3
$2y$13$gZ4dqdbumUfTxgXQXOo7e.jKJpMKFMa90YzwhimNyZY...	admin	admin	a
$2y$13$kNNmizgEl4hNcy3vench/.b65eYx4PHtUOiNCH977tU...	Deroy	Flavien	5
$2y$13$eBSeGu1CZVKG4b.QJFT.eeLB/OMwMxonNUnhgm39AjZ...	BOURGEOIS	Agnes	1

CREATION D'UN DEVIS

On entame ainsi la deuxième partie la plus essentielle du projet qui est la création d'un devis à partir d'un tarif calculé.

Pour commencer, on calcule ce qu'on récupère dans cette page :



Ainsi que le type client du compte Utilisateur afin de déterminer si le calcul doit détenir une promotion ou non. On récupère donc cela dans le controller DevisController :

```
        $choix= $session->get('choix');
```

```
        if (!empty($this->getUser())) {
            $mail = $this->getUser()->getUserIdentifier();
            $monUser = new User();
            $monUser = $doctrine->getRepository(User::class)-
>findOneBy(array('email' => $mail));

            $typeClient = $monUser->getTypeClient();
        } else {
            $typeClient = '';
        }
```

Et les données du formulaire ci-dessus :

```
        $haie=$request->get('haie');
        $hauteur=$request->get('hauteur');
        $longueur=$request->get('longueur');
        $prix = $doctrine->getRepository(Haie::class)->find($haie)->getPrix();
        $maHaie = $doctrine->getRepository(Haie::class)->find($haie);
```

Le devis doit être calculé de la sorte :

Prix = Prix unitaire(*) x Longueur haie
Si Hauteur > 1m50, multiplier le prix par 1.5
Si l'utilisateur est une entreprise, appliquer une remise de 10%
Le calcul est basé ainsi sur le prix du type de haie choisie.

Et j'ai procédé au calcul comme ceci :

```
{% if maHaie.getNom == "Abélia" %}
    {% set prixhaie = prix * longueur %}
{% elseif maHaie.getNom == "Laurier" %}
    {% set prixhaie = prix * longueur %}
{% elseif maHaie.getNom == "Thuya" %}
    {% set prixhaie = prix * longueur %}
{% elseif maHaie.getNom == "Troène"%}
    {% set prixhaie = prix * longueur %}
{% endif %}

{% if hauteur > 150 %}
    {% set prixWithHauteur = prixhaie * 1.5 %}
{% else %}
    {% set prixWithHauteur = prixhaie * 1 %}
{% endif %}

{% if typeClient == 'e' %}
    {% set pourcentremise = 10 %}
    {% set remise = prixWithHauteur * 0.10 %}
    {% set calculTotal = prixWithHauteur - remise %}
    <br>
    <p>Vous avez obtenu une remise de :
        {{remise}}€</p>
{% else %}
    {% set pourcentremise = 0 %}
    {% set remise = 0 %}
    {% set calculTotal = prixWithHauteur %}
{% endif %}


{% if typeClient == 'p' %}
    {% set choixlabel = 'Particulier' %}
{% elseif typeClient == 'e' %}
    {% set choixlabel = 'Entreprise' %}
{% endif %}

<p>Vous êtes un(e)
    {{choixlabel}}
    (vous bénéficiez d'une remise de
    {{pourcentremise}}%).
    <br>
```

```
                    Rappel : Mesures de la haie : Longueur
                    {{longueur}}m Hauteur
                    {{hauteur}}m
                    <br>
                    Le montant de votre devis est de :
                    {{calculTotal}}€</p>
```

Ce qui donne un récapitulatif et qui donne la possibilité donc de générer le devis en gardant les données de type de haie, longueur et largeur.

Récapitulatif de votre devis.
Vous avez obtenu une remise de : 12€
Vous êtes un(e) Entreprise (vous bénéficiez d'une remise de 10%).
Rappel : Mesures de la haie : Longueur 12m Hauteur 12m
Le montant de votre devis est de : 108€
Vous souhaitez créer ce devis ?
Créer le devis

La création se fait à l'aide getters ainsi que de setters à l'aide de données de sessions :

```
$choix = $session->get('choix');             // On utilise les get pour obtenir les données de l'id de l'input voulue
        $haie = $session->get('haie');
        $hauteur = $session->get('hauteur');
        $longueur = $session->get('longueur');

        if (!empty($this->getUser())) {
            $mail = $this->getUser()->getUserIdentifier();
            $monUser = new User();
            $monUser = $doctrine->getRepository(User::class)->findOneBy(array('email' => $mail));

            $typeClient = $monUser->getTypeClient();
        } else {
            $typeClient = '';
        }

        $codehaie = $doctrine->getRepository(Haie::class)->find($haie);

        $devis = new Devis();
```

```php
        $devis->setUser($monUser);
        $devis->setHaie($codehaie);
        $devis->setLongueur($longueur);
        $devis->setHauteur($hauteur);
        $devis->setDate(new \DateTime());

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

GESTION D'UN DEVIS

Après avoir créé son devis, il faut évidemment avoir la possibilité de consulter ses devis créés ou s'il on veut, les supprimer.

Pour cela, et pour les autres gestions d'utilisateurs et des haies, j'ai utilisé le système de création de contrôleurs CRUD automatique à partir d'une entité Doctrine ORM.

Pour créer un CRUD :

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

```
The class name of the entity to create CRUD (e.g. GrumpyGnome):
> Haie

Choose a name for your controller class (e.g. HaieController) [HaieController]:
> HaieController

created: src/Controller/HaieController.php
created: src/Form/HaieType.php
created: templates/haie/_delete_form.html.twig
created: templates/haie/_form.html.twig
created: templates/haie/edit.html.twig
created: templates/haie/index.html.twig
created: templates/haie/new.html.twig
created: templates/haie/show.html.twig
```

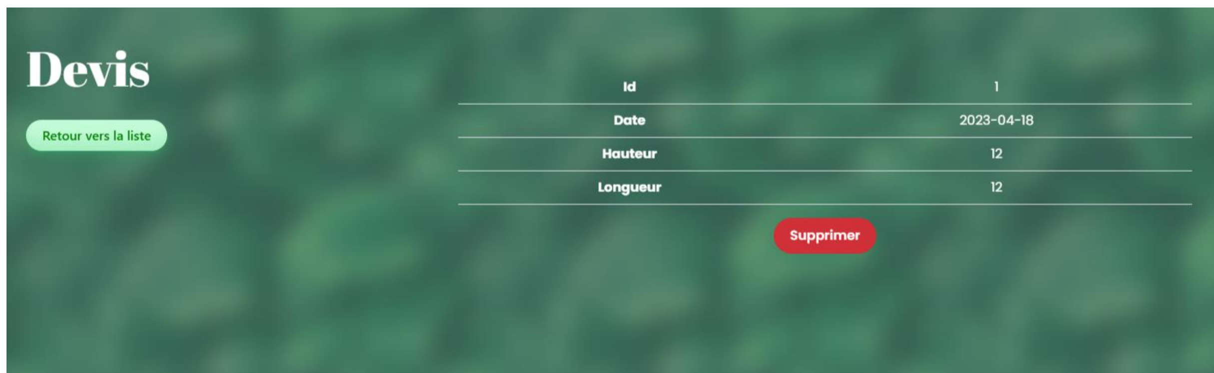
```
Next: Check your new CRUD by going to /haie/
```

Ce qui va donner un total de six dossiers qui ont chacun leur utilité (modifier, lire, supprimer)

```
devis_crud
  _delete_form.html.twig
  _form.html.twig
  edit.html.twig
  index.html.twig
  new.html.twig
  show.html.twig
```

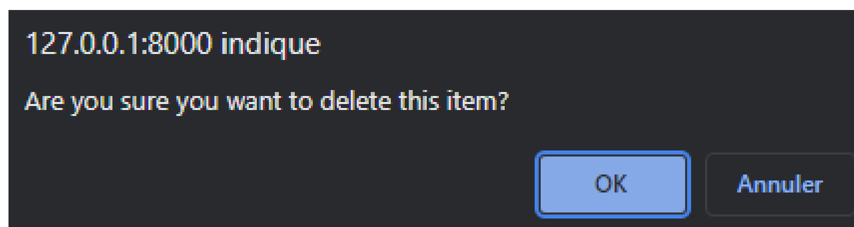
C'est grâce à ce système que j'ai pu générer automatiquement toutes les pages qui permettent de consulter et autres.

Liste grâce au fichier devis_crud/index.html.twig comportant la liste de tous les devis



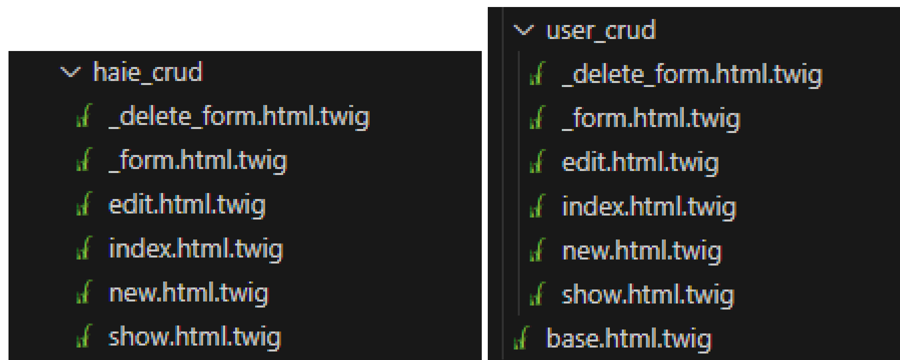
Ensemble des informations du devis grâce à la page devis_crud/show.html.twig

La page devis_crud/_delete_form.html.twig ne détient évidemment pas de page spécifique à la suppression mais apporte un pop-up permettant de confirmer son choix cela dit.



GESTION DES UTILISATEURS ET DES HAIES (ADMINISTRATEURS)

Comme j'ai pu le citer précédemment, cette partie n'a pas été très compliqué mais plutôt rapide car j'utilise le composant existant créant des CRUD selon des entités. Dans ce cas là j'ai créé donc des CRUD pour la User et Haie que j'ai relié dans les fonctionnalités pour l'administrateur à l'aide des routes :




```
        {% if is_granted('ROLE_ADMIN') %}
            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" href="#"
id="navbarDropdown" role="button" data-bs-toggle="dropdown" data-
toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                    Gestion des haies
                </a>
                <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
                    <li>
                        <a class="dropdown-item"
href="{{path('app_haie_crud_index')}}">Liste des haies</a>
                    </li>
                    <li>
                        <a class="dropdown-item"
href="{{path('app_haie_crud_new')}}">Ajouter une haie</a>
                    </li>
                </ul>
            </li>

            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" href="#"
id="navbarDropdown" role="button" data-bs-toggle="dropdown" aria-
expanded="false">
                    Gestion des clients
                </a>
                <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
                    <li>
```

```
                        <a class="dropdown-item"
href="{{path('app_user_crud_index')}}">Consulter la liste des clients</a>
                    </li>
                    </li>
                </ul>
            </li>
        {% endif %}
```

Cela a donc permis l’ajout de ces pages :

Création d’une nouvelle haie directement dans l’application grâce au fichier haie_crud/new.html.twig

Liste grâce au fichier haie_crud/index.html.twig comportant la liste de toutes les haies

Liste des clients inscrits

Id	Email	Roles	Nom	Prenom	Adresse	Ville	Code Postal	Actions
1	jules.artaud@outlook.com	["ROLE_USER"]	Jules	Artaud	309 Rue François Perrin	Limoges	87000	Afficher Modifier
2	admin@admin.fr	["ROLE_ADMIN","ROLE_USER"]	admin	admin	admin	admin	admin	Afficher Modifier
3	deroy.flavien@gmail.com	["ROLE_USER"]	Deroy	Flavien	5 Rue de Rochechouart	Limoges	87000	Afficher Modifier
4	agnes.bourgeois@ac-limoges.fr	["ROLE_USER"]	BOURGEOIS	Agnes	11 Rue de Valadon	Limoges	87000	Afficher Modifier

Liste grâce au fichier user_crud/index.html.twig comportant la liste de tous les utilisateurs inscrits

TESTS ET VERIFICATION

Pour remarquer si les travaux sont bien fonctionnels comme il se doit en respectant les obligations nécessaires, la phase de test est primordiale.

Pour se faire il est obligatoire de tester chaque méthode du CRUD générés par Symfony ainsi que la conception élaborée soi-même.

Les tests sont évidemment validés si les données s’implémentent bien avec les caractéristiques demandées en base de données.

CONCLUSION

Travailler avec le Framework Symfony m'a ouvert les yeux sur la dynamique et la rapidité du développement, offrant ainsi un outil extrêmement efficace. J'ai pu apprécier ses nombreux avantages, tels que sa structure solide, sa flexibilité et sa documentation complète. Ce projet m'a permis d'acquérir de nouvelles compétences et de m'adapter à différents contextes d'entreprise. De plus, j'ai pris plaisir à ajouter ma touche personnelle, en particulier en améliorant mes compétences en CSS, Bootstrap et PHP.