

AP3.2 : Un projet PHP avec le Framework Symfony



PRÉSENTATION

Présentation de l'équipe et du projet

Durée du projet	Du 17/12/2021 au 25/03/2022
Qui sommes-nous?	L'équipe est composée de deux élèves de 2ème année en BTS SIO option SLAM. Christian et Mélanie.

OBJECTIFS

Afin de réaliser ce projet, nous avons plusieurs tâches à accomplir :

- **Compléter le menu de l'application** : ajouter la gestion des clients et la gestion des devis
- **Gestion des clients** : recherche/ajout/modification/suppression des clients
- **Gestion des devis** : calcul du devis en fonction du type de client (particulier ou entreprise) pour obtenir la remise et des différents types de haie du devis (hauteur et longueur)

Compléter le menu de l'application

Pour rajouter la gestion des clients et des devis nous avons juste eu à modifier le fichier **base.html.twig** qui correspond à l'interface du site « **Le P'tit Jardinier** ».

```
<li class="nav-item active">
```

```
<a class="nav-link" href="{{ path('accueil') }}">
    Accueil
</a>
</li>
<li class="nav-item dropdown">
    <a class="nav-link" href="{{ path('choix') }}">
        Faire un devis en ligne
    </a>
</li>
<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown"
    role="button" data-toggle="dropdown" aria-haspopup="true" aria-
    expanded="false">
        Gestion des types de haie
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
        <a class="nav-link" href="{{ path('creerHaie') }}">
            Création
        </a>
        <a class="nav-link" href="{{ path('rechercher') }}">
            Consultation/Modification
```



```

<li class="nav-item dropdown">

    <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown"
    role="button" data-toggle="dropdown" aria-haspopup="true" aria-
    expanded="false">

        Gestion des devis

    </a>

    <div class="dropdown-menu" aria-labelledby="navbarDropdown">

        <a class="nav-link" href="{{ path('app_devis') }}">

            Création d'un devis

        </a>

        <a class="nav-link" href="{{ path('app_consultation_devis') }}">

            Consultation/Modification d'un devis

        </a>

    </div>

</li>

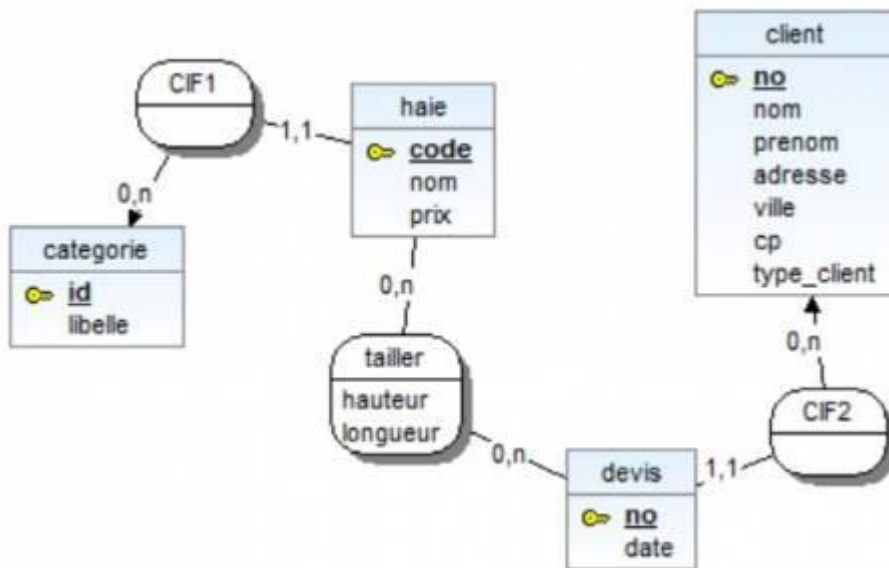
```

Résultat sur le site :



Gestion des clients

Avant de commencer la gestion des clients nous avons dû intégrer la capture du MCD dans la base de données stockée sur **phpMyAdmin**.



Pour rajouter la table **client** nous avons utilisé la commande **php bin/console make:entity**.

Afin de gérer les types de client de la même façon qu'on avait géré les catégories précédemment, nous avons rajouté une table **type_client**, et nous avons lié à la table **client**.

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

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

```
> typeClient
```

```
Your entity already exists! So let's add some new fields!
```

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

```
> client
```

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

```
> relation
```

```
What class should this entity be related to?:
```

```
> Client
```

```
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
```

```
> ManyToOne
```

Is the typeClient.client property allowed to be null (nullable)? (yes/no) [yes]:

> no

Do you want to add a new property to Client so that you can access/update Haie objects from it - e.g. \$client->getTypeClient ()? (yes/no) [yes]:

>

A new property will also be added to the Client class so that you can access the related typeClient objects from it.

New field name inside Client [devis]:

>

Do you want to activate orphanRemoval on your relationship?

A typeClient is "orphaned" when it is removed from its related Client.

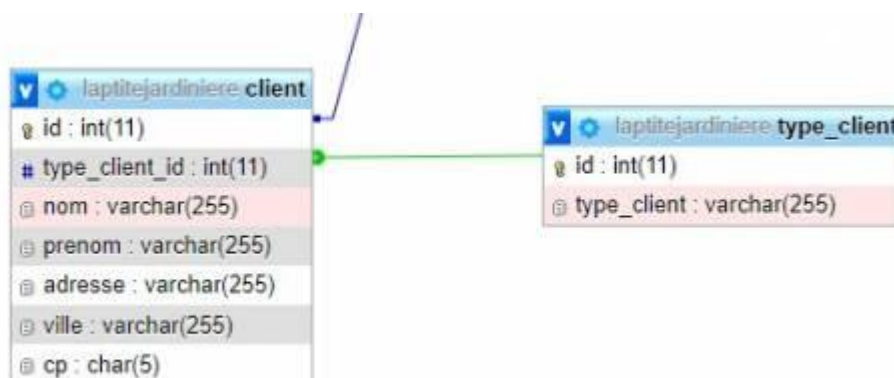
e.g. \$client->removetypeClient (\$typeClient)

NOTE: If a typeClient may *change* from one Client to another, answer "no".

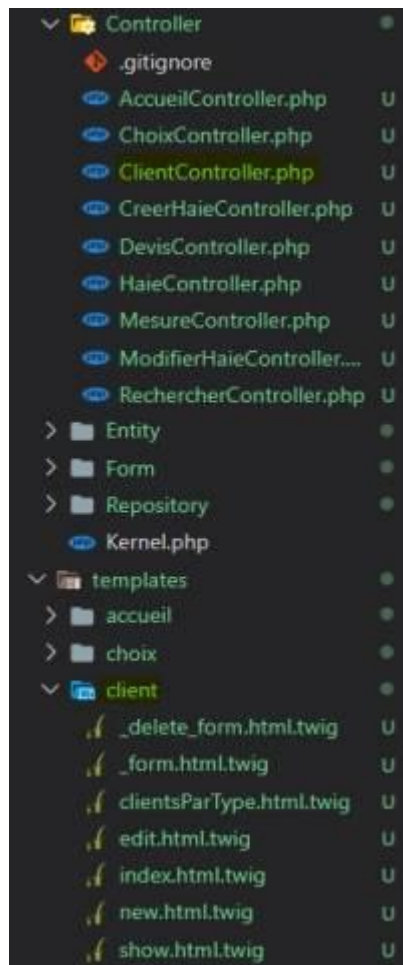
Do you want to automatically delete orphaned App\Entity\typeClient objects (orphanRemoval)? (yes/no) [no]:

> no

Résultat du concepteur sur PhpMyAdmin :



Pour gérer les clients, nous avons la commande **php bin/console make:crud** qui permet de créer automatiquement les formulaires d'ajout, de modification, de suppression et d'affichage pour une entité.



Création d'un client ☒

Grâce à l'élément **CRUD**, plusieurs fichiers, dont la fonction d'insertion ont été automatiquement créés. Néanmoins, nous avons dû modifier le fichier **ClientType** de la **class Form** afin qu'une liste déroulante remplis des deux types de client (particulier et entreprise) s'affiche.

```
->add('typeClient', EntityType::class, [ 'class' => TypeClient::class,
'choice_label' => 'type_client']]);
```

Ensuite, pour éviter l'erreur : **L' objet de la classe Vapp\Entity\ n'a pas pu être converti en chaîne**, nous avons ajouté la méthode **toString** dans la classe **TypeClient** qui renvoie **type_client** qui est le nom de l'état qui sera affiché dans le **select**.

```
public function __toString() {

    return $this->type_client;

}
```

Après une modification du **templates de l'ajout**, nous pouvons voir à quoi ressemble l'ajout sur l'application.

— AJOUTER UN CLIENT —

Nom :	Prénom :
<input type="text" value="Entrez le nom"/>	<input type="text" value="Entrez le prénom"/>
Adresse :	Ville :
<input type="text" value="Entrez l'adresse"/>	<input type="text" value="Entrez la ville"/>
Code Postal :	Type de client :
<input type="text" value="Entrez le code postal"/>	<input type="text" value="Particulier"/>

Lorsque nous procédons à l'ajout d'un client, nous voulons qu'un message d'alerte apparaisse pour confirmer à l'utilisateur que l'ajout a bien été réalisé.

Pour réaliser cela, nous avons rajouté l'élément **addFlash** dans la fonction d'ajout.

```
$this->addFlash('success', 'Le client a bien été créé !');
```

Ensuite dans le fichier **index.html.twig** qui permet d'afficher le tableau ci-dessus, nous avons rajouté une condition qui permet d'afficher le message.

```
{% for message in app.flashes('success') %}  
  
<div class="alert alert-success">  
  
    {{ message }}  
  
</div>  
  
{% endfor %}
```

Résultat :

———— CONSULTATION ————

Le client a bien été créé !

Nom	Prenom	Adresse	Ville	Cp	Type de client	Choix	
Bogusz	Thierry	Valadon	Limoges	87000	Entreprise	Modifier	Supprimer
Bourgeois	Agnès	Valadon	Limoges	87000	Entreprise	Modifier	Supprimer
toto	toto	totodresse	totoville	87100	Particulier	Modifier	Supprimer
Pasqualini	Claude	test	Saint-Junien	87200	Entreprise	Modifier	Supprimer

Comme nous pouvons le voir ci-dessus, dans le tableau se trouve deux boutons concernant la modification et la suppression. Cela nous amène au point suivant.

Modification/Suppression d'un client ✂

Comme indiqué précédemment, grâce à l'élément **CRUD**, les fonctions de modification et de suppression d'un client se sont créées automatiquement. Nous n'avons pas eu à les modifier.

Après une modification du **templates de modification**, nous pouvons voir à quoi ressemble la modification sur l'application.

———— MODIFIER UN CLIENT ————

Nom :

Prénom :

Adresse :

Ville :

Code Postal :

Type de client :

Lorsque nous validons la modification du client sélectionné, nous sommes renvoyés sur le tableau de consultation et nous pouvons y voir nos modifications, ainsi qu'un message d'alerte nous indiquant que la/les modification(s) a/ont bien été pris en compte.



CONSULTATION

Le client a bien été modifié !

Nom	Prenom	Adresse	Ville	Cp	Type de client	Choix	
Bogusz	Thierry	Valadon	Limoges	87000	Entreprise	Modifier	Supprimer
Bourgeois	Agnès	Valadon	Limoges	87000	Entreprise	Modifier	Supprimer
toto	toto	totodresse	totoville	87100	Particulier	Modifier	Supprimer
Pasqualini	Claude	en France	Saint-Junien	87200	Entreprise	Modifier	Supprimer

Pour réaliser cela, nous avons repris ce que nous avons fait pour l'ajout et nous avons dupliqué ça pour la suppression.

La suppression se gère de manière simple grâce à la fonction **delete**.



CONSULTATION

Le client a bien été supprimé !

Nom	Prenom	Adresse	Ville	Cp	Type de client	Choix	
Bogusz	Thierry	Valadon	Limoges	87000	Entreprise	Modifier	Supprimer
Bourgeois	Agnès	Valadon	Limoges	87000	Entreprise	Modifier	Supprimer
toto	toto	totodresse	totoville	87100	Particulier	Modifier	Supprimer

Recherche d'un client 🔍

Pour gérer la **recherche d'un client** nous avons créé un nouveau **controller** (**SearchController**), afin de créer deux fonctions de recherche. Ne sachant pas selon quoi la recherche devra se faire, nous en avons déduit qu'elle se faisait selon le **nom** et le **prénom** du client.

Nous avons créé une **function** qui nous avons nommé **searchBar**. Ce formulaire sera affiché dans la sidebar.

```
public function searchBar()

{

    $form = $this->createFormBuilder()

        ->setAction($this->generateUrl('handleSearch'))

        ->add('query', TextType::class, [

            'label' => false,

            'attr' => [

                'class' => 'form-control',

                'placeholder' => 'Rechercher (Nom et/ou Prénom)'

            ]

        ])

        ->add('recherche', SubmitType::class, [

            'attr' => [

                'class' => 'btn btn-success'

            ]

        ])

        ->getForm();

    return $this->render('search/searchBar.html.twig', [

        'form' => $form->createView()

    ]);
}
```

```
}
```

Nous avons un champ **query (TextType)** qui est le champ où l'utilisateur pourra entrer le nom et/ou prénom du client et un bouton recherche (**SubmitType**).

Ensuite, dans ce **controller** nous avons géré la requête. Il s'agit de la partie importante et la plus complexe.

```
/**
 * @Route("/searchClient", name="searchClient")
 * @param Request $request
 */
public function searchClient(Request $request, ClientRepository $repo)
{
    $query = $request->request->get('form')['query'];

    if($query) {
        $clients = $repo->findClientsByName($query);
    }

    return $this->render('client/searchClient.html.twig', [
        'clients' => $clients
    ]);
}
```

Nous avons rajouté une **@Route(handleSearch)** qui va s'occuper de récupérer la **function findClientsByName()** dans le **ClientRepository** et va **render** le templates **search/index.html.twig**.

Dans le **ClientRepository.php**, nous avons créé une **function findClientsByName()** qui ira chercher les noms et prénoms des clients.

```
public function findClientsByName($query)
{
```

```

$qb = $this->createQueryBuilder('c');

$qb

    ->where(

        $qb->expr()->andX(

            $qb->expr()->orX(

                $qb->expr()->like('c.nom', ':query'),

                $qb->expr()->like('c.prenom', ':query'),

            ),

        )

    )

    ->setParameter('query', '%' . $query . '%');

return $qb

    ->getQuery()

    ->getResult();

}

```

Ensuite, dans le templates **searchBar.thml.twig**, nous avons seulement rajouté ceci :

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

Et c'est le templates **searchClient.thml.twig** qui aura la partie la plus importante.

```

{% if app.request.method == 'POST' %}

{% if clients | length == 0 %}

</br>

<h4 style="text-align:center">Aucun client trouvé.</h4>

{% else %}

```

```
<table class="table table-hover" style="width: 55%;text-align:center;
position:relative;left:350px">
```

```
    <thead class="thead-dark">
```

```
        <tr>
```

```
            <th>
```

```
                Nom
```

```
            </th>
```

```
            <th>
```

```
                Prenom
```

```
            </th>
```

```
            <th>
```

```
                Adresse
```

```
            </th>
```

```
            <th>
```

```
                Ville
```

```
            </th>
```

```
            <th>
```

```
                Cp
```

```
            </th>
```

```
            <th>
```

```
                Type de client
```

```
            </th>
```

```
        </tr>
```

```
    </thead>
```

```
    <tbody>
```

```
{% for client in clients %}

<tr>

    <td>

        {{ client.nom }}

    </td>

    <td>

        {{ client.prenom }}

    </td>

    <td>

        {{ client.adresse }}

    </td>

    <td>

        {{ client.ville }}

    </td>

    <td>

        {{ client.cp }}

    </td>

    <td>

        {{ client.typeClient }}

    </td>

</tr>

{% endfor %}

{% endif %}

{% endif %}
```

Résultat sur le site :



RECHERCHE DES CLIENTS

Rechercher (Nom et/ou Prénom)

Recherche

Si nous rentrons la lettre **e** dans le champ de recherche, nous aurons pour résultat tous les clients ayant cette lettre dans leur prénom et/ou nom.



RECHERCHE DES CLIENTS

Rechercher (Nom et/ou Prénom)

Recherche

Nom	Prenom	Adresse	Ville	Cp	Type de client
Bogusz	Thierry	Valadon	Limoges	87000	Entreprise
Bourgeois	Agnès	Valadon	Limoges	87000	Entreprise
Pasqualini	Claude	chez lui	Saint-Junien	87200	Entreprise

Gestion des devis

Avant de commencer la gestion des devis nous avons dû intégrer la suite du MCD dans la base de données stockée sur **phpMyAdmin**.

Pour rajouter la table devis nous avons utilisé la commande **php bin/console make:entity**.

Pour la relation **tailler** qui doit devenir une table, nous l'avons géré de manière un peu différente à l'aide de **Mme Bourgeois**.

En effet, nous avons tenté de gérer l'association **Tailler** avec la relation **ManyToMany** mais ça ne fonctionnait pas car **Symfony** fait du modèle objet et nous, avec **Doctrine** nous faisons du mapping relationnel. En effet, nous avons un modèle objet et une base de données relationnelle. Malheureusement, **Symfony** n'arrive pas à bien le gérer et donc ne fait pas apparaître l'association **Tailler** dans notre modèle objet.

C'est pourquoi nous avons détourné le problème.

Nous avons créé la table **Tailler** de manière classique (avec la commande **php bin/console make:entity**). Nous y avons inséré les propriétés hauteur et longueur. Puis, pour faire le lien avec les tables **Devis** et **Haie**, nous l'avons lié avec la relation **ManyToOne**.

Résultat :



Afin de rendre les clés composées (**haie_id**, **devis_id**) unique, nous avons rajouté une contrainte dans l'entité **Tailler**, nommé **UniqueConstraint**.

```
/**  
  
 * @ORM\Table(  
  
 * name="tailler",  
  
 * uniqueConstraints={  
  
 * @ORM\UniqueConstraint(name="association_unique", columns={"haie", "devis"  
 * })}  
 * }  
 */  
  
class Tailler  
  
{
```

Pour gérer la **gestion des devis** c'est-à-dire le calcul du devis en fonction du type de client (particulier ou entreprise) pour obtenir la remise et des différents types de haie du devis (**hauteur** et **longueur**) nous avons dû modifier plusieurs **Controller** ainsi que leurs **templates** que nous avons créés pour la première étape de l'application.

Modification du **ChoixController** :

```
public function index(TypeClientRepository $TypeClientRepository): Response {  
  
    $typeClient = $TypeClientRepository->findAll();  
  
    return $this->render('choix/index.html.twig', [  
  
        'controller_name' => 'ChoixController',  
  
        'typeClient' => $typeClient  
  
    ]);  
  
}
```

Modification de son **templates choix** :

```
<select class="form-control" name="choix" id="choix">  
  
    <option value="">  
  
        Choisir...  
  
    </option>  
  
    {% for item in typeClient %}  
  
    <option>  
  
        {{ item.getTypeClient() }}  
  
    </option>  
  
    {% endfor %}  
  
</select>
```

Avant, les types de clients étaient mis au dur mais à présent ils sont liés à la base de données.

Modification du **resultDevisController** :

```
public function index(): Response
{
    $session = new Session();

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

    $request = Request::createFromGlobals();

    $mesure = $request->get('mesure');

    $longueur = $request->get('longueur');

    $hauteur = $request->get('hauteur');

    $prixHaie = 0;

    switch ($mesure) {
        case "Laurier":
            $prixHaie = 30;

            break;

        case "Thuya":
            $prixHaie = 35;

            break;

        case "Troène":
            $prixHaie = 28;

            break;

        case "Abélia":
            $prixHaie = 25;
```

```

        break;

    }

    $total = $prixHaie * $longueur;

    if ($hauteur > 1.5) {

        $total = $total * 1.5;

    }

    $montantRemise = 0;

    $remise = 0;

    if ($choix == "Entreprise") {

        $remise = 10;

        $montantRemise = $total / 10;

        $total = $total - $montantRemise;

    }

    return $this->render(

        'resultDevis/index.html.twig',

        array(

            'choix' => $choix, 'longueur' => $longueur, 'hauteur' =>
$hauteur, 'mesure' => $mesure, 'remise' => $remise, 'montantRemise' =>
$montantRemise, 'total' => $total

        )

    );

}

```

Modification de son **templates result_devis** :

```
<label for="exampleFormControlSelect1" style="font-size:25px">
```

```
<B>Détail du devis</B>
```

```
</br>
```

Vous êtes

```
{{ choix }}
```

```
{% if (choix == 'Entreprise') %}
```

(vous bénéficiez d'une remise de 10%)

```
{% endif %}
```

```
</br></br>
```

```
<B>Information de la haie :</B>
```

```
</br>
```

Haie sélectionnée : {{ mesure }}

```
</br>
```

Longueur de la haie : {{ longueur }} m

```
</br>
```

Hauteur de la haie : {{ hauteur }} m

```
</br></br>
```

```
<B>Résultat du devis :</B>
```

```
{% if (choix == 'Entreprise') %}
```

```
</br>
```

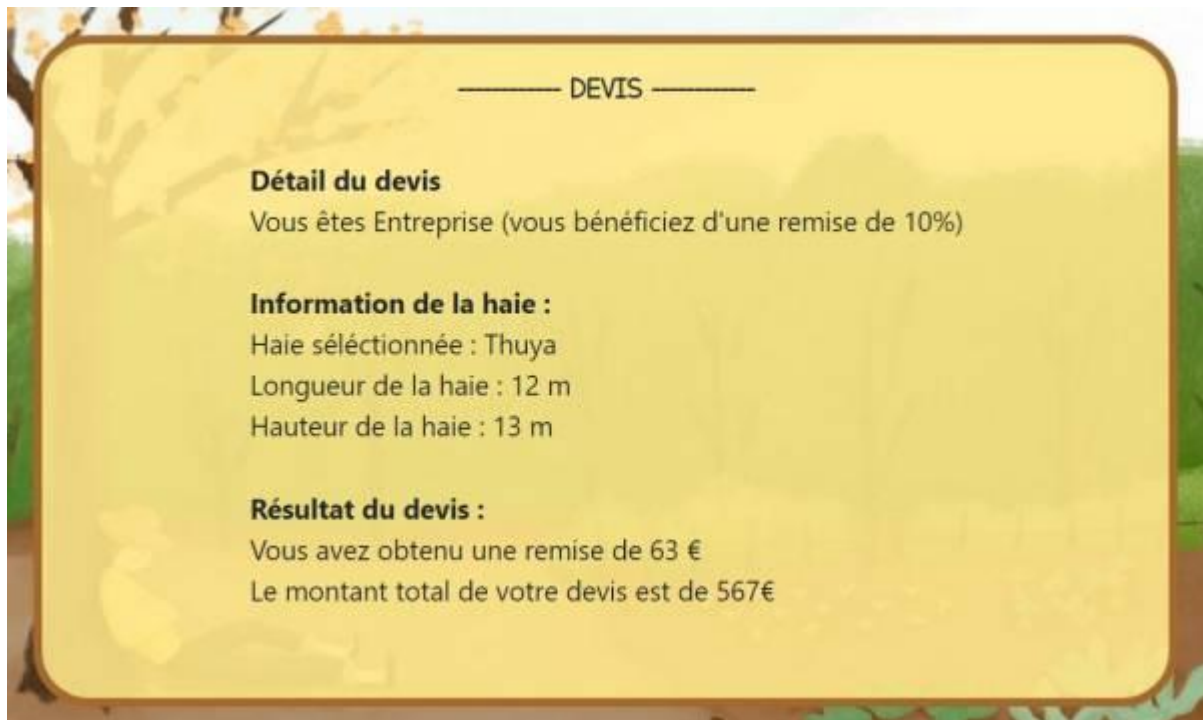
Vous avez obtenu une remise de

```
{{ montantRemise }} €
```

```
{% endif %}  
  
</br>  
  
Le montant total de votre devis est de  
  
{{ total }}€  
  
</label>
```

Comme nous pouvons le voir ci-dessus, les conditions ne sont plus gérées dans le **templates** mais dans le **contrôleur**.

À présent nous pouvons voir le **détail du devis** avec le **montant total** :



Gérer l'ajout, l'affichage et la modification des devis

Pour gérer la création du devis nous avons dû créer deux contrôleurs (**DevisController** et **TaillerController**) car la création d'un devis passe par l'insertion dans deux tables (devis et tailler).

Création d'un devis ☒

Nous avons commencé par la création du contrôleur **DevisController**. L'insertion dans la table **devis** devra prendre en compte la date du jour et le client.

```
public function newDevis(Request $request, EntityManagerInterface  
$entityManager): Response
```

```

{

    $devi = new Devis();

    $devi -> setDate(new \DateTime('now'));

    $forms = $this->createForm(DevisType::class, $devi);

    $forms->handleRequest($request);

    if ($forms->isSubmitted() && $forms->isValid()) {

        $entityManager->persist($devi);

        $entityManager->flush();

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

    }

    return $this->renderForm('devis/index.html.twig', [

        'devi' => $devi,

        'forms' => $forms

    ]);

}

```

Comme nous pouvons le voir ci-dessus, dans les premières lignes de la fonction, nous appelons le fichier **devisType**.

En effet, ce dernier va nous permettre de faire afficher une liste déroulante remplis des noms et prénoms des clients.

```

->add('client', EntityType::class, ['label' => 'Client : ', 'class' =>
Client::class, 'choice_label' => 'PrenomAndNom', 'attr' => ['class' => 'form-
control',]]);

```

Mais, afin que le nom et le prénom s'affiche dans la même liste déroulante, nous avons créé une fonction dans la **class Client** pour les liés, nommé **PrenomAndNom()**.

```
public function getPrenomAndNom() {  
  
    return $this->prenom . ' ' . $this->nom;  
  
}
```

Ensuite, pour éviter l'erreur : **L' objet de la classe App\Entity\ n'a pas pu être converti en chaîne** , nous avons ajouté la méthode **toString** dans la classe **Client** qui renvoie de nouveau le **nom** et le **prénom** du client.

```
public function __toString() {  
  
    return $this->prenom . ' ' . $this->nom;  
  
}
```

Nous n'avons plus qu'à modifier le templates du devis afin d'y intégrer tout ça.

```
{{ form_label(forms.client, 'Choisissez un client : ' ) }}  
  
{{ form_widget(forms.client, {'attr':{'class':'form-control' }} ) }}
```

A présent, nous pouvons voir à quoi ressemble la première partie de l'ajout d'un devis.



Ensuite, nous nous sommes occupés de la création du controller **TaillerController**. L'insertion dans la table **tailler** devra prendre en compte la haie, la hauteur, la longueur et le dernier id du devis qu'on aura créé.

```
public function newTailler(Request $request, EntityManagerInterface  
$entityManager): Response
```



```

{

    $tailleur = new Tailleur();

    $form = $this->createForm(TailleurType::class, $tailleur);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {

        $tailleur->setDevis($devis);

        $entityManager->persist($tailleur);

        $entityManager->flush();

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

    }

    return $this->renderForm('tailleur/index.html.twig', [

        'tailleur' => $tailleur,

        'form' => $form

    ]);

}

```

Comme nous pouvons le voir ci-dessus, dans les premières lignes de la fonction, nous appelons le fichier **tailleurType**.

En effet, ce dernier va nous permettre de faire afficher une liste déroulante remplis des noms des haies et des champs hauteur et longueur ainsi que la gestion des erreurs.

```

->add('longueur', NumberType::class, array('invalid_message'=>'Vous devez
saisir un nombre !'))

```

```

->add('hauteur', NumberType::class, array('invalid_message'=>'Vous
devez saisir un nombre !'))

->add('haie', EntityType::class, ['label' => 'Haie', 'class' =>
Haie::class, 'choice_label' => 'nom']);

```

Ensuite, pour éviter l'erreur : ***L' objet de la classe App\Entity n'a pas pu être converti en chaîne***, nous avons ajouté la méthode **toString** dans la classe **Haie** qui renvoie de nouveau le **nom** de la haie.

```

public function __toString()

{

    return $this->nom;

}

```

Nous n'avons plus qu'à modifier le **templates** du devis afin d'y intégrer tout ça.

```

<td>

    {{ form_label(form.haie, 'Choisissez votre type de haie : ' ) }} {{
form_widget(form.haie, {'attr':{'class':'form-control' }} ) }}

</td>


<td>

    {{ form_label(form.longueur, 'Longueur (en mètre) :' ) }} {{
form_widget(form.longueur, {'attr':{'class':'form-control', 'placeholder' :
'Entrez la longueur (en mètre)' }} ) }}

</td>


<td>

    {{ form_label(form.hauteur, 'Hauteur (en mètre) :' ) }} {{
form_widget(form.hauteur, {'attr':{'class':'form-control', 'placeholder' :
'Entrez la hauteur (en mètre)' }} ) }}

</td>

```

Néanmoins, pour qu'à chaque ajout dans la table **tailler** nous récupèrerions le dernier id créé de la table **devis**, nous avons modifié les controllers **DevisController**, et **TaillerController**.

Modification du **DevisController** :

Nous avons rajouté ceci dans le `if ($forms→isSubmitted() && $forms→isValid())`.

Cela nous permet d'envoyer le dernier idDevis créé vers le controller Tailler.

```
$session = new Session();  
  
$session->set("idDevis", $devi->getId());
```

Modification du **TaillerController** :

Nous avons rajouté ceci dans le `if ($forms→isSubmitted() && $forms→isValid())`.

Ici nous récupérons dans une variable de session l'idDevis envoyé précédemment.

```
$session = new Session();  
  
$idDevis = $session->get("idDevis");  
  
$devis = $this->getDoctrine()  
    ->getRepository(Devis::class)  
    ->find($idDevis);  
  
$tailler->setDevis($devis);
```

A présent, nous pouvons voir à quoi ressemble la première partie de l'ajout d'un devis dans la table **tailler**.



Lorsque nous procédons à l'ajout d'un devis, nous voulions laisser à l'utilisateur la possibilité **d'insérer plusieurs haie dans un devis**, mais aussi la possibilité de le/les consulter.

Pour réaliser cela, nous avons rebouclé sur la page d'ajout des haies puis nous avons ajouté à l'élément **addFlash**, la méthode **generateUrl** qui permet de générer un lien vers un nouvel URL.

```
$newPageUrl = $this->generateUrl('app_consultation_devis');

$this->addFlash(

    'success',

    sprintf('Si vous voulez finir votre devis, <a href="%s">cliquer
ici ! </a> </br> </br> Vous pouvez également poursuivre votre sélection ci-
dessous', $newPageUrl)

);
```

Résultat sur le site :

Si l'utilisateur clique sur le lien de consultation, il devra arriver vers une nouvelle page répertoriant tous les devis créés.

Pour réaliser cela, nous avons créé un nouveau contrôleur : **ConsultationDevisController** où nous avons simplement fait un `findAll` de tailler.

```
$tailleur = $tailleurRepository->findAll();
```

Modification du templates de la consultation :

```
<tr>

  <th>

    Date

  </th>

  <th>

    Client

  </th>

  <th>

    Haie

  </th>

  <th>

    Hauteur

  </th>

  <th>

    Longueur

  </th>

  <th colspan="2" rowspan="2">

    Choix

  </th>

</tr>

</thead>

<tbody>

  {% for taille in tailler %}
```

```

<tr>

    <td>


    </td>

    <td>

        {{ taille.devis }}

    </td>

    <td>

        {{ taille.haie }}

    </td>

    <td>

        {{ taille.hauteur }} m

    </td>

    <td>

        {{ taille.longueur }} m

    </td>

    <td>

        <a type="button" class="btn btn-warning" href="{{
path('app_modif_devis_tailler', {'id': taille.id}) }}">

            Modifier

        </a>

    </td>

    <td>

        <form method="post" action="{{ path('tailler_devis_delete', {'id':
taille.id}) }}" onsubmit="return confirm('Souhaitez-vous vraiment supprimer ce
devis ?');">

```

```

        <input type="hidden" name="_token" value="{{
csrf_token('delete' ~ taille.id) }}">

        <button class="btn btn-danger">

            Supprimer

        </button>

    </form>

</td>

</tr>

{% endfor %}

```

Résultat sur le site :



Date	Client	Haie	Hauteur	Longueur	Choix
18/03/2022	Claude Pasqualini	Thuya	55 m	59 m	Modifier Supprimer
19/03/2022	Agnès Bourgeois	Laurier	5 m	5 m	Modifier Supprimer
20/03/2022	Thierry Bogusz	Troène	55 m	5 m	Modifier Supprimer
22/03/2022	Claude Pasqualini	Troène	28 m	32 m	Modifier Supprimer

Comme nous pouvons le voir ci-dessus, dans le tableau se trouve deux boutons concernant la modification et la suppression. Cela nous amène au point suivant.

Modification/Suppression d'un devis ✂

Ajout de la fonction de modification d'un devis :

```

public function modif(Tailler $tailler, Request $request,
EntityManagerInterface $em)

{

```

```

        $forms = $this->createForm(TaillerDevisType::class, $tailler);

        $forms->handleRequest($request);

        if ($forms->isSubmitted() && $forms->isValid()) {

            $em->persist($tailler);

            $em->flush();

            $this->addFlash('success', 'Le devis a bien été modifié !');

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

        }

        return $this->render(

            'tailler/modif.html.twig',

            array('forms' => $forms->createView())

        );

    }

```

Nous voulions que la modification se fasse sur le même formulaire contrairement à l'ajout. Pour cela, nous avons créé un second form **TaillerDevisType** où nous appelons le form DevisType (il contient le client).

```

->add('longueur', NumberType::class, array('invalid_message'=>'Vous devez saisir un nombre !'))

->add('hauteur', NumberType::class, array('invalid_message'=>'Vous devez saisir un nombre !'))

->add('haie', EntityType::class, ['label' => 'Haie', 'class' => Haie::class, 'choice_label' => 'nom'])

->add('devis', DevisType::class);

```


Modification de templates afin d'y inclure la modification.

Résultat sur le site :



The screenshot shows a web form titled "MODIFIER UN DEVIS". It contains four input fields: "Client :" with a dropdown menu showing "Claude Pasqualini", "Haie :" with a dropdown menu showing "Abélia", "Longueur :" with a text input showing "5", and "Hauteur :" with a text input showing "5". At the bottom right, there are two buttons: "Modifier le devis !" (green) and "Annuler" (red).

Lorsque nous validons la modification du devis sélectionné, nous sommes renvoyés sur le tableau de consultation et nous pouvons y voir nos modifications, ainsi qu'un message d'alerte nous indiquant que la/les modification(s) a/ont bien été pris en compte.



The screenshot shows a web page titled "CONSULTATION DES DEVIS". At the top, there is a message box that says "Le devis a bien été modifié !". Below this is a table with the following data:

Date	Client	Haie	Hauteur	Longueur	Choix	
18/03/2022	Claude Pasqualini	Thuya	55 m	59 m	Modifier	Supprimer
19/03/2022	Agnès Bourgeois	Laurier	5 m	5 m	Modifier	Supprimer
20/03/2022	Thierry Bogusz	Troène	55 m	5 m	Modifier	Supprimer
22/03/2022	Christian Sun	Troène	28 m	32 m	Modifier	Supprimer

Pour réaliser cela, nous avons repris de nouveau ce que nous avons fait pour le client.

La suppression se gère de manière simple grâce à la fonction **delete**.

Comme précédemment, un message d'alerte apparaît lorsque la suppression a bien été effectuée.



Information

La table **client** est lié à la table **devis** qui est lui-même lié à la table **tailler**. De ce fait, comme à présent nous gérons le devis, lorsque nous voulons supprimer un client, un erreur apparait indiquant que nous ne pouvons pas le supprimer car sa **clé étrangère** est présente dans les devis existant.

Afin de gérer au mieux cette situation nous avons fait en sorte que lorsque nous voulons supprimer un client nous supprimons également les devis dépendant de ce client.

Pour cela, nous avons rajouté cet élément dans la **class Client** et dans la **class Devis**.

```
cascade={"remove"})
```

Mais, afin d'informer l'utilisateur de cela, nous avons modifié l'alerte de confirmation de suppression.

```
<form method="post" action="{ path('client_delete', {'id': client.id}) }"
onsubmit="return confirm('ATTENTION !!! Si vous supprimer ce client, vous
risquez de supprimer également les devis qui dépendent de ce client.');">
```

```
    <input type="hidden" name="_token" value="{ csrf_token('delete' ~
client.id) }" >
```

```
    <button class="btn btn-danger">
```

```
        Supprimer
```

```
    </button>
```

```
</form>
```

A présent si nous cliquons sur le bouton **Supprimer** afin de supprimer un client, voilà ce que apparait :



Donc, si nous cliquons sur **OK**, le client est supprimé ainsi que le/les devis dépendant de lui.

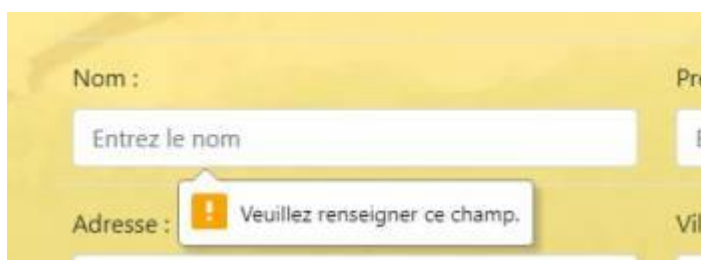
Système de gestion des erreurs

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.

Résultat :



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",

*      maxMessage = "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 !'))

```

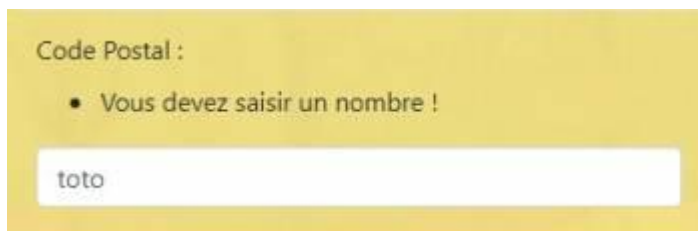
Pour que ce message d'erreur apparaisse, nous avons rajouté dans le templates **_form.html.twig** cette petite phrase de code.

```

{{ form_errors(form.cp) }}

```

Résultat :



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

