Symfony CRM – Gestion Clients & Factures

Ce projet est une application CRM (*Customer Relationship Management*) développée avec Symfony 7. Il permet à un utilisateur de gérer ses propres clients et les factures associées, le tout dans une interface épurée et fonctionnelle. Il inclut une chaîne CI/CD complète avec Jenkins, Docker, SonarQube et Ansible.

I. Étapes de mise en place

1. Création du projet Symfony

composer create-project symfony/skeleton CRM

2. Création des entités avec relations

**Utilisateur** (lié à un client via une relation OneToMany).

**Client** (lié à un utilisateur via une relation ManyToOne).

**Facture** (lié à un client via une relation ManyToOne).

II. Bonnes Pratiques

1. Validation des données

Utilisation des contraintes

**Assert\NotBlank** : Vérifie que le champ **n’est pas vide**.

**Assert\Length** : Définit une **longueur minimale et/ou maximale** pour une chaîne de caractères.

**Assert\Regex** : Vérifie que la valeur **respecte une expression régulière** (regex).

**UniqueEntity** : Vérifie qu’une valeur est **unique dans la base de données**.

Validation automatique lors de la soumission des formulaires.

2. Ajout des formulaires avec Symfony FormType

Création des ClientType et FactureType

Utilisation de form\_row() dans Twig pour une intégration propre

**buildForm()**

**Rôle :** Définit les **champs du formulaire** (nom, type, label, options, contraintes, etc.)

**configureOptions()**

**Rôle :** Définit les **options globales du formulaire**, en particulier l’entité à laquelle le formulaire est lié (data\_class).

1. ****Séparation des responsabilités (principe SOLID)****

Le FormType contient **la logique de construction du formulaire**.

Le contrôleur reste concentré sur la logique métier.

**S = Single Responsibility Principle**

2. ****Réutilisables****

Tu peux utiliser le même FormType :

pour l’ajout, l’édition et dans différents contrôleurs

3. ****Centralisation de la configuration****

Labels, required, attr, constraints (validation), etc., sont définis **à un seul endroit**.

4. ****Intégration directe avec les entités****

Tu peux lier directement un FormType à une entité (data\_class), ce qui permet :

le **mappage automatique des données**

la gestion facile des objets (création, modification)

5. ****Support des contraintes de validation****

Symfony applique automatiquement les **Assert** de l’entité au moment de la soumission.

6. ****Gestion simple des types complexes****

Tu peux facilement intégrer des relations (EntityType, CollectionType, etc.)

7. DRY Dont Repeate Yourself

* Au lieu d’écrire les champs de formulaire à la main dans chaque vue ou contrôleur
* La **structure du formulaire est définie une seule fois** dans ClientType et réutilisée partout : create, edit, etc.

5. Authentification sécurisée

1. Configuration via security.yaml

Security

provider :

indique à Symfony comment charger les utilisateurs depuis la base de données.  
Ici, Symfony va chercher dans la table utilisateur avec la colonne identifiant.

security:

providers:

app\_user\_provider:

entity:

class: App\Entity\Utilisateur

property: identifiant

password\_hashers :

Symfony va hasher les mots de passe automatiquement selon la meilleure méthode.

password\_hashers:

Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'

Firewall :

Le firewall dev désactive la sécurité pour les ressources du debug et les assets (CSS, JS, etc.).

Le firewall main protège les pages du site.

|  |  |
| --- | --- |
| lazy: true | Charge l'utilisateur uniquement si nécessaire (meilleure perf) |
| provider: | Utilise app\_user\_provider pour authentifier |
| custom\_authenticator: | Utilise ta classe LoginFormAuthenticator |
| logout: | Permet à l’utilisateur de se déconnecter |
| remember\_me: | Active la case à cocher “Se souvenir de moi” (7 jours ici) |

access\_control:

**Contrôle d’accès** : qui peut accéder à quelles URLs.

| URL | Autorisation requise |
| --- | --- |
| /login | Accès public |
| / (page d’accueil) | Accès public |
| Toutes les autres (^/) | Nécessite ROLE\_USER |

2. Utilisation de UserInterface et PasswordAuthenticatedUserInterface

**UserInterface :**

Cette interface est obligatoire pour toute classe utilisateur en Symfony.

Elle définit les méthodes de base nécessaires à l’authentification :

* public function getUserIdentifier(): string;
* public function getRoles(): array;
* public function eraseCredentials(): void;

Elle permet à Symfony de reconnaître l’utilisateur connecté et de gérer les rôles.

Cette interface est utilisée pour **indiquer que l'utilisateur possède un mot de passe** (et que Symfony doit le comparer via le PasswordHasher).

Elle ajoute la méthode :

* public function getPassword(): ?string;

3. Routes protégées avec isGranted () et getUser()

**getUser() :**

Permet d’**accéder à l’utilisateur actuellement connecté** dans un contrôleur Symfony :

* $utilisateur = $this->getUser();

Tu peux ensuite filtrer les données :

$clients = $clientRepository->findBy(['utilisateurOwner' => $utilisateur]);

**isGranted() :**

Permet de **vérifier si l’utilisateur a un rôle donné**, ou s’il a le droit d’effectuer une action.

**Exemples :** Vérification dans un contrôleur

if (!$this->isGranted('ROLE\_ADMIN')) {

throw $this->createAccessDeniedException("Accès refusé.");

}

**Exemples :** Vérification dans Twig

{% if is\_granted('ROLE\_ADMIN') %}

<a href="{{ path('admin\_dashboard') }}">Admin</a>

{% endif %}

6. Messages Flash

Le projet utilise les **messages flash de Symfony** pour **informer l’utilisateur après certaines actions**, comme :

* ✅ Ajout d’un client
* ❌ Suppression ou erreur

Autorisations fines avec Voter et isGranted ()

L'application utilise des **Voters personnalisés** pour gérer les droits d'accès aux entités sensibles, comme les clients. Cela permet d'appliquer des règles comme :

"Un utilisateur ne peut modifier ou supprimer qu’un client dont il est propriétaire."

Mise en place du Voter

Le fichier ClientVoter.php contient les règles d'autorisation pour les actions suivantes :

CLIENT\_VIEW

CLIENT\_EDIT

CLIENT\_DELETE

**La méthode supports ()**

**Rôle :** Détermine si **se voter doit s’appliquer ou non** selon :

l'action demandée (attribut : ex. CLIENT\_EDIT)

l’objet (ex. une instance de Client)

**La méthode voteOnAttribute ()**

**Rôle :** Contient la **vraie logique de permission** :

Est-ce que cet utilisateur a le droit d’effectuer cette action sur cet objet ?

Utilisation dans les contrôleurs

Exemple dans ClientController :

$this->denyAccessUnlessGranted('CLIENT\_DELETE', $client);

Cela déclenche automatiquement le ClientVoter pour vérifier si l'utilisateur est propriétaire du client.

Utilisation dans les vues Twig

Dans les templates (ex. client/index.html.twig), les actions sensibles sont affichées uniquement si l’utilisateur est autorisé :

{% if is\_granted('CLIENT\_EDIT', client) %}

<a href="{{ path('client\_edit', {'id': client.id}) }}">Modifier</a>

{% endif %}

{% if is\_granted('CLIENT\_DELETE', client) %}

<form action="{{ path('client\_delete', {'id': client.id}) }}" method="post">

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

<button class="btn btn-danger">Supprimer</button>

</form>

{% endif %}

8. Utilisation de Bootstrap pour les vues Twig

Séparation logique des vues : client/index.html.twig, facture/edit.html.twig...

Gestion des événements : Connexion utilisateur (LoginSubscriber)

L'application utilise un **EventSubscriber** pour **intercepter la connexion réussie d’un utilisateur** et afficher un message flash personnalisé.

Objectif

Afficher un message du type :

Bienvenue Moncef !

juste après la connexion.

Implémentation

Fichier : src/EventSubscriber/LoginSubscriber.php

<?php

namespace App\EventSubscriber;

use App\Entity\Utilisateur;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

use Symfony\Component\HttpFoundation\Session\Session;

use Symfony\Component\HttpFoundation\Session\SessionInterface;

use Symfony\Component\Security\Http\Event\LoginSuccessEvent;

class LoginSubscriber implements EventSubscriberInterface

{

private Session $session;

public function \_\_construct(SessionInterface $session)

{

if (!$session instanceof Session) {

throw new \LogicException('Expected Session instance');

}

$this->session = $session;

}

public static function getSubscribedEvents(): array

{

return [

LoginSuccessEvent::class => 'onLoginSuccess',

];

}

public function onLoginSuccess(LoginSuccessEvent $event): void

{

$user = $event->getUser();

if ($user instanceof Utilisateur) {

$this->session->getFlashBag()->add('success', 'Bienvenue ' . $user->getNom() . ' !');

}

}

}

9. Affichage dans le template

Dans base.html.twig, ajouter après la <nav> :

<div class="container mt-3">

{% for label, messages in app.flashes %}

{% for message in messages %}

<div class="alert alert-{{ label }} alert-dismissible fade show" role="alert">

{{message}}

<button type="button" class="btn-close" data-bs-dismiss="alert"></button>

</div>

{% endfor %}

{% endfor %}

</div>

Bonnes pratiques utilisées

1. SOLID

**S - Single Responsibility Principle**

Chaque classe a une responsabilité claire (ex : ClientController = gérer les clients).

**O - Open/Closed Principle**

Code extensible via les entités et formulaires, sans modifier la logique des contrôleurs.

**L - Liskov Substitution Principle**

Les objets (Utilisateur, Client, etc.) respectent les contrats attendus.

**I - Interface Segregation Principle**

Interfaces fines utilisées (UserInterface, PasswordAuthenticatedUserInterface).

**D - Dependency Inversion Principle**

Injection de dépendances via les **constructeurs**, en utilisant EntityManagerInterface, ClientRepository, etc.

Design Patterns appliqués

****1. MVC (Model - View - Controller)****

Architecture native de Symfony : entités = Modèle, Controller = logique, Twig = Vue.

****2. Dependency Injection****

Services injectés automatiquement avec autowire dans les contrôleurs et services.

****3. Repository Pattern****

Utilisation des repositories comme couche d'accès aux données (ClientRepository, FactureRepository).

Déploiement avec Jenkins + Ansible

Étapes du pipeline (dans Jenkinsfile)

Clonage du dépôt GitLab

Installation avec Composer

Nettoyage du cache Symfony

Analyse du code avec SonarQube

Construction de l’image Docker

Push de l’image vers DockerHub

Déploiement automatisé via Ansible

Structure du projet

devops\_crm/

├── ansible/

│ ├── deploy.yml

│ └── inventory.ini

├── code/ # Symfony app

├── docker-compose.yml

├── infra/

│ └── php/

│ └── Dockerfile

├── jenkins/

│ └── Jenkinsfile

└── README.md