Dossier du projet "Réaliser un site e-commerce avec Symfony"

Table des matières

A. Présentation

1. Contexte

La marque de sweat-shirt Stubborn souhaite se doter d'une boutique en ligne pour que les clients puissent acheter des sweat-shirts.

2. Les objectifs du site

Le site doit permettre aux utilisateurs de se créer un compte. Une fois ce compte créé ils pourront accéder à la boutique, sélectionner un sweat-shirt pour accéder à sa fiche détaillée. Sur cette page ils pourront sélectionner une tailler et l'ajouter au panier. Ainsi ils pourront consulter leur panier sur une page dédiée où il leur sera possible de supprimer un article ou de payer le/les article(s). Le paiement se fera par carte bleue.

Sur ce site il existe un compte administrateur qui ouvre l'accès à la page back-office. Sur cette page il est possible d'ajouter un sweat-shirt ainsi que de modifier/supprimer les sweat-shirts existants.

3. Périmètre du projet

Ce projet est développé à l'aide du framework Symfony tant pour l'interface front-end que pour l'interface back-end.
Les informations seront enregistrées dans une base de données relationnelle grâce à l'ORM Doctrine embarqué par Symfony qui permet de faire le pont entre l'application et le SGBD MySQL.

Le projet se limite au développement de l'interface et des fonctionnalités métiers qui découlent de l'activité d'un site e-commerce. De cette façon, les styles et mises en forme seront minimes et non traités dans ce projet.

On peut distinguer 3 statuts d'interaction de l'utilisateur avec l'application : non connecté, connecté en tant que client et connecté en tant qu'administrateur.

Lorsqu' il n'est pas connecté, l'utilisateur a accès uniquement aux pages d'accueil, d'inscription et de login.
Une fois connecté en tant que client, il a accès à toutes les pages du site à l'exception de la page back-office.
Lorsqu'il est connecté en tant qu'administrateur, l'utilisateur a alors accès à l'intégralité des pages de l'application.

Lors de l'inscription, un mail de vérification est envoyé sur l'adresse mail de l'utilisateur pour valider son compte afin qu'il puisse se connecter avec ses identifiants. Ce processus d'envoie de mail utilise le bundle symfony/mailer.

Une fois le panier rempli, le client peut valider sa commande en cliquant sur le bouton "FINALISER MA COMMANDE" qui ouvre une modale où apparaît l'interface fournie stripe qui permet de payer avec la carte bleue.

Sur l'interface back-office, il est possible d'ajouter un nouveau sweat, pour cela il faut une photo, un nom, un prix et un stock d'au moins 2 exemplaires pour chaque taille. On peut aussi renseigner le fait que le pull apparaisse ou non sur la page d'accueil du site.
Une fois le sweat ajouté, il est possible de modifier toute ces informations relatives au sweat et il est aussi possible de le supprimer de la base de données.

B. Graphisme et ergonomie

1. *La charte graphique*

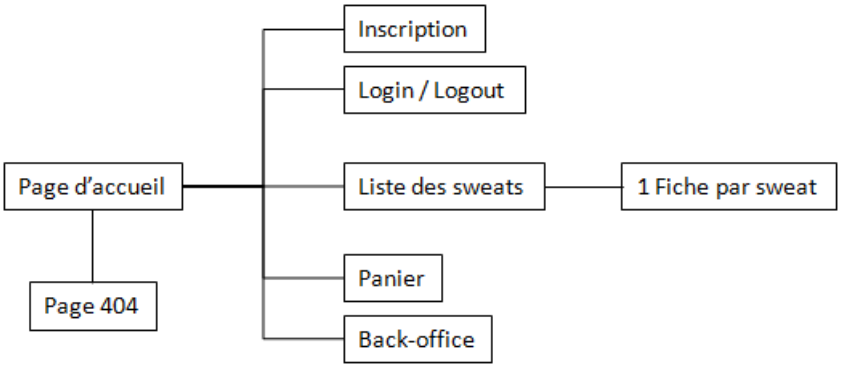
La charte graphique n'est pas encore mise en place par le client. Seule la favicon a été choisi par le client :

Cette même image est utilisée sur l'en-tête de chaque page pour retourner à tout moment sur la page d'accueil.

2. *Maquettage*

Le client préfère que les fonctionnalités soient d'abord développées pour s'assurer que leur utilisation lui convient avant de les intégrer dans des maquettes. Le maquettage sera donc réalisé par la suite et prendra en compte les différentes fonctionnalités mises en place au cours de ce projet.

3. *Arborescence du site web*



C. Spécificités et livrable

1. *Le contenu du site*

a. Page d'accueil

En tête

L'en-tête contient le logo de l'entreprise qui permet de retourner à la page d'accueil en cliquant dessus. On y trouve aussi le menu de navigation qui est variable selon le statut de connexion de l'utilisateur.

Utilisateur non connecté :

- Accueil
- Inscription
- Connexion

Utilisateur connecté en tant que client :

- Accueil
- Boutique
- Panier
- Déconnexion

Utilisateur connecté en tant qu'administrateur :

- Accueil
- Boutique
- Panier
- Back-office
- Déconnexion

Par défaut, cet en-tête sera présent sur toutes les pages de la plateforme.

Corps de la page

Il affiche les cards des sweats que l'administrateur a choisi de mettre en évidence sur son site.
Sur ces cards, on peut voir la photo du sweat, son nom et son prix. De plus lorsque l'utilisateur est connecté il y a un bouton "VOIR" sur ces cards qui permettent d'accéder à la ficher détaillée du sweat concerné.

Pied de page

Un encadré contenant une description de l'entreprise Stubborn.

Par défaut, ce pied de page sera présent sur toutes les pages de la plateforme.

b. Page d'inscription

Cette page contient le formulaire d'inscription avec les champs :

- Nom d'utilisateur
- Adresse mail
- Adresse de livraison
- Mot de passe
- Confirmer le mot de passe

Tous les champs sont requis pour que l'inscription soit valide.

Un lien "Se Connecter" est présent pour que l'utilisateur se rende sur la page de login s'il possède déjà un compte.

c. Page de login

Cette page contient le formulaire de connexion avec les champs :

- Nom d'utilisateur
- Mot de passe

Un lien 'Mot de passe oublié' est présent mais sans fonctionnalité pour le moment
Un lien "Créer un compte" est présent pour que l'utilisateur puisse se rendre sur la page d'inscription s'il ne possède pas encore de compte.

d. Page des produits

Cette page affiche une card par sweat et cela pour l'ensemble des sweats que vend l'entreprise. Il y a aussi un filtre permettant d'afficher les sweats selon leur prix :

- 10 à 30 €

- 30 à 35 €
- 35 à 50 €

Chaque card présente :
- Une photo du sweat
- Le nom du sweat
- Le prix du sweat
- Un lien "VOIR" pour accéder à la ficher détaillée du sweat

e. Page d'un produit spécifique

Cette page doit présenter une photo du sweat, son nom et son prix. De plus il doit être possible de sélectionner la taille du sweat (XS, S, M, L ou XL) et de l'ajouter au panier.

f. Page du panier

Il doit y avoir un récapitulatif de tous les sweats présent dans le panier. Pour chacun d'eux on doit y retrouver la photo, le nom, le prix, la taille sélectionnée et la possibilité de le retirer du panier.

A la fin du récapitulatif, on doit trouver le prix total du panier, un lien "CONTINUER MES ACHATS" pour retourner sur la boutique ainsi qu'un bouton "FINALISER MA COMMANDE". Lorsque l'utilisateur clique sur ce dernier, une modale doit s'ouvrir et présenter un formulaire où l'utilisateur peut renseigner sa carte bleue et payer.

g. Page back-office

Cette page doit présenter un premier formulaire permettant de renseigner :
- une photo du sweat,
- le nom du sweat,
- son prix,
- le stock pour chacun des tailles (XS, S, M, L, XL) avec un minimum de 2 exemplaires pour chaque taille,
- s'il apparaît ou non sur la page d'accueil.

On doit trouver ce même formulaire pour chacun des sweat déjà présent en base de données pour pouvoir modifier chacune des informations du sweat ou de pouvoir le supprimer totalement de la base de données.

h. Page 404

L'utilisateur est redirigé sur cette page dès que le chemin de l'URL n'est pas reconnu. L'utilisateur est alors invité à revenir sur la page d'accueil via un lien "Retour à la page d'accueil".

2. *Spécificité techniques*

a. Création d'un compte

Parlons du formulaire d'inscription pour imager l'utilisation des formulaires dans Symfony.

La création d'un compte se fait sur la page dont le chemin est '/register'. La requête pour cette route est gérée par le contrôleur RegistrationController.php.
Tout d'abord il faut commencer par définir la route dans le contrôleur et le nom qui lui est associé :

```
#[Route('/register', name: 'app_register')]
```

Puis la fonction juste en dessous va gérer les requêtes GET et POST de ce chemin.
Pour les requêtes GET et POST on va commencer par définir une nouvelle instance d'une entité Customer, cette instance va contenir notamment des propriétés comme $email, $password, $name qui sont en faite les noms des différents attributs de la table Customer en base de données. Cette nouvelle instance va ensuite être incorporée dans une instance de FormInterface générée à partir de notre classe RegistrationFormType définissant les champs du formulaire d'inscription. Dans notre class RegistrationFormType on définit les champs du formulaire avec les mêmes noms que ceux des propriétés de notre Customer.
De cette façon lorsqu'on renseigne notre utilisateur dans notre nouvelle instance de FormInterfase générée à partir de notre RegistrationFormType, on a une association qui se fait directement entre les propriétés de notre Customer et les champs du formulaire.

```
$user = new Customer();
$form = $this->createForm(RegistrationFormType::class, $user);
$form->handleRequest($request);
```

Pour les requêtes GET on peut alors directement passer la méthode qui va permettre de réaliser le rendu et qui prend en premier argument le chemin vers le template twig depuis le dossier templates et en second argument un tableau associatif qui va transmettre au template les variables que l'on souhaite utiliser dans le template.

```
return $this->render('registration/register.html.twig', [
    'registrationForm' => $form,
]);
```

Là on va pouvoir accéder au contenu de $form via registrationForm dans le template. Cela va notamment permettre de générer le formulaire :

```
{{ form_start(registrationForm, {
    'attr': {'class': 'registration-form'}
}) }}…
{{ form_end(registrationForm) }}
```

Il est possible de faire appel à chaque champs du formulaire contenu dans registrationForm (donc dans $form) et même d'y ajouter des attributs comme les classes par exemple :

```
{{ form_label(registrationForm.email, null, {'label_attr': {'class': 'registration-form__email-label'}}) }}
{{ form_widget(registrationForm.email, {'attr': {'class': 'registration-form__email-input'}}) }}
```

Une fois que l'utilisateur a rempli ce formulaire, il faire une requête POST toujours sur l'URL contenant le pathname('/register').
De la même façon que pour une requête GET, une nouvelle instance de Customer est placée dans une nouvelle instance de FormInterface sur le modèle de notre RegistrationFormType.
Mais cette fois la méthode handleRequest() va venir complété les propriétés de notre $user avec le contenu des champs renseignés par l'utilisateur.

Ensuite on rentre dans un bloc conditionnel grâce à la méthode isSubmitted() qui renvoie true puisque le formulaire est soumis. On vérifie alors tout d'abord que les deux passwords renseignés par l'utilisateur sont identiques et si ce n'est pas le cas on génère une instance de FormError ce qui invalide le formulaire :

```
if ($form->isSubmitted()) {
    /** @var string $plainPassword */
    $plainPassword = $form->get('plainPassword')->getData();
    $confirmPassword = $form->get('confirm_password')->getData();

    // Check that password and confirmed password are the same
    if ($plainPassword !== null && $confirmPassword !== null && $plainPassword !== $confirmPassword) {
        $form->get('confirm_password')->addError(new FormError('Les mots de passe ne sont pas identiques.'));
    }
```

Dans le cas où les deux passwords sont identiques et peut tester la condition $form->isValid() et rentrer dans le bloc conditionnel si c'est le cas. Dans ce cas, on va d'abord hash le password afin qu'il ne soit pas lisible en base de données puis il suffit d'utiliser la méthode $entityManager->persist($user) pour annoncer à Doctrine que l'on souhaiter ajouter le $user à la base de donnée en tant qu'un enregistrement de la table Customer (puisque $user est une instance de la class Customer). Puis on utilise la méthode $entityManager->flush() pour valider note demande à Doctrine.

```
// Form submission
if ($form->isValid()) {
    // encode the plain password
    $user->setPassword($userPasswordHasher->hashPassword($user, $plainPassword));

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

Ensuite on envoie un mail grâce au bundle symfony/mailer qui est utilisé par EmailVerifier (cette partie est développée ci-dessous).

Enfin on redirige l'utilisateur vers la page d'accueil avec le paramètre de requête registered=1 pour afficher un message spécifique sur la page d'accueil :

```
return $this->redirectToRoute('app_home', ['registered' => 1]);
```

b. Mailer

Comme vu ci-dessus. Le mail de vérification est envoyé au moment de l'inscription juste après l'enregistrement en base de données du nouvel utilisateur. Pour cela on utilise la méthode sendEmailConfirmation() de la classe EmailVerifier qui prend en premier argument le nom qui fait référence à la route qui va gérer la vérification de l'email, ici pour nous c'est 'app_verify_email'. Le second argument est l'instance de l'utilisateur à qui on envoie le mail et le troisième une instance de la classe TemplatedEmail. Puis on peut enchainer les méthodes :

- from : qui prend en argument notre adresse mail,
- to : qui prend en argument l'adresse mail de l'utilisateur,
- subject : qui prend en argument le sujet du mail,
- htmlTemplate : qui prend en argument le chemin du template à utiliser pour le mail,
- context : qui prend en argument un tableau associatif pour transmettre une variable au template (utilisée ici pour personnaliser le mail),
- locale : qui prend en argument la langue à utiliser dans le template

De cette façon l'utilisateur reçoit l'email de confirmation contenant un lien pour activer son compte. Ce lien va effectuer une requête à l'URL dont la chemin est '/verify/email' :

```
#[Route('/verify/email', name: 'app_verify_email')]
```

On récupère alors l'identifiant de l'utilisateur présent dans une query string de l'URL pour récupérer l'utilisateur en base de données.

```
$id = $request->query->get('id');
if (!$id === null) {
    $this->addFlash('verify_email_error', "Le compte a été supprimé de notre base
    return $this->redirectToRoute('app_home');
}

$user = $customerRepository->find($id);
if (!$user === null) {
    $this->addFlash('verify_email_error', "Cette adresse mail n'est pas reconnue
    return $this->redirectToRoute('app_home');
```

De cette façon on peut utiliser la méthode handleEmailConfirmation() de la classe EmailVerifier qui prend en argument la requête et l'utilisateur. Cela va changer en base de donne la valeur de l'attribut is_verified de 0 (false) à 1 (true) pour que l'utilisateur puisse utiliser ces identifiants afin de se connecter à la base de données.

```php
// validate email confirmation link, sets User::isVerified=true and persists
try {
    $this->emailVerifier->handleEmailConfirmation($request, $user);
} catch (VerifyEmailExceptionInterface $exception) {
    $this->addFlash('verify_email_error', $translator->trans($exception->getReason(), [], 'VerifyEmailBundle', 'fr'));

    return $this->redirectToRoute('app_home');
}
```

Enfin si la vérification s'est bien déroulée, on ajoute un flashes à notre instance via $this->addFlash(). Ces flashes sont une sorte de 'localStorage' propre à Symfony où l'information est disponible à partir de la prochaine requête puis dès qu'ils sont affichés, ils sont supprimés.
On redirige alors l'utilisateur vers la page d'accueil qui affichera le flashes transmis.

```php
$this->addFlash('success', 'Merci, votre adresse mail a bien été validée. Votre compte est
return $this->redirectToRoute('app_home');
```

c. Paiement Stripe

Commençons du côté front-end où il faut tout d'abord charger le script js de stripe grâce à la balise :

```html
<script src="https://js.stripe.com/v3/"></script>
```

Puis dans notre fichier js, on récupère notre clé publique fournit par le back-end lors du rendu de la page et stocké dans une div :

```html
<div id='stripe-public-key' data-public-key="{{ publicKey }}"></div>
```

```javascript
const publicKeyElement = document.getElementById('stripe-public-key');
const publicKey = publicKeyElement.dataset.publicKey;
```

Ensuite on initialise stripe avec notre clé publique grâce à la fonction Stripe() fournit par Stripe. Cela va ensuite nous permettre de générer un élément stripe dans lequel on va incorporer un formulaire de carte bleue fournit par Stripe. Enfin on peut monter l'élément dans l'élément HTML que l'on souhaite :

```javascript
const stripe = Stripe(publicKey);
const elements = stripe.elements();
const cardElement = elements.create("card");
cardElement.mount("#card-element");
```

Maintenant que le formulaire est accessible au client, il peut le remplir et cliquer sur le bouton 'Payer'. Cela va envoyer une requête fetch à notre serveur sur l'url avec le chemin '/card-payment'

```javascript
try {
    const response = await fetch('/cart-payment');
```

De cette façon côté serveur, on commence par calculer le montant total du panier :

```php
#[Route('/cart-payment', name:'app_cart_payment')]
#[IsGranted('ROLE_USER')]
public function cartPayment(CartRepository $cartRepository, StripeService $stripeService): JsonResponse
{
    $cartProducts = $cartRepository->findWithSweatAndSizeByCustomer($this->getUser());

    // Set amount to pay
    $totalPrice = 0;
    foreach ($cartProducts as $cartProduct) {
        $totalPrice += $cartProduct->getSweatVariant()->getSweat()->getPrice();
    }
    $amount = (int) round($totalPrice*100); // $amount is in cents
```

Puis on génère une demande de paiement (paymentIntent) via notre service Stripe :

```php
$paymentIntent = $stripeService->createPaymentIntent($amount);
```

Dans le service stripe on utilise la classe PaymentIntent fournie par le bundle de stripe pour générer cette intention de paiement qui va être enregistrée sur notre compte Stripe.

```php
class StripeService
{

    public function __construct()
    {
        Stripe::setApiKey($_ENV['STRIPE_SECRET_KEY']);
    }

    public function createPaymentIntent(int $amount, string $currency = 'eur'): PaymentIntent
    {
        return PaymentIntent::create([
            'amount' => $amount, // amount in cents
            'currency' => $currency,
            'automatic_payment_methods' => ['enabled' => true],
        ]);
    }
}
```

Ensuite notre contrôleur renvoie une réponse au format json au client qui contient une clé secrète destinée au client, qui est spécifique à l'intention de paiement générée.

```php
return new JsonResponse([
    'clientSecret' => $paymentIntent->client_secret,
]);
```

Côté client, on récupère la clé secrète du client et on utilise la méthode asynchrone confirmCardPayment() qui prend en premier argument la clé secrète du client pour retrouver l'intention de paiement côté serveur Stripe. Le second argument qui est un objet contenant les informations de la carte fournie par le client et son nom pour retrouver à qui correspond la facture du côté de notre compte stripe.

```
try {
    const response = await fetch('/cart-payment');
    const { clientSecret } = await response.json();
    const { error, paymentIntent } = await stripe.confirmCardPayment(clientSecret, {
        payment_method: {
            card: cardElement,
            billing_details: {
                name: userName
            }
        }
    })
```

Si le paiement est un succès alors le client envoie une requête fetch à l'url contenant le chemin '/cart-payment-confirmed' :

```
if (paymentIntent && paymentIntent.status == "succeeded") {
    const response = await fetch('/cart-payment-confirmed');
```

De cette façon côté serveur, on peut récupérer tous les éléments contenus dans le panier, les supprimer du panier et les stocker dans une entité qui s'appellerait OrderEntity afin de conserver une trace de l'achat par exemple (cette entité n'a pas encore été créée mais c'est une piste d'amélioration de l'application). Ensuite on ajoute un flashes de succès pour le client et on retourne une réponse json :

```
#[Route('/cart-payment-confirmed', name:'app-cart-payment-confirmed')]
#[IsGranted('ROLE_USER')]
public function cartPaymentConfirmed(CartRepository $cartRepository, EntityManagerInterface $em): JsonResponse
{
    // Empty the cart now products have been payed
    $cartProducts = $cartRepository->findBy(['customer' => $this->getUser()]);
    foreach ($cartProducts as $cartProduct) {
        $em->remove($cartProduct);
        $em->flush();
        // Next step is to create an Order Entity to save the orders
    }
    $this->addFlash('success', "Le paiement est validé, nous vous remercions pour votre achat");
    return new JsonResponse([
        'success' => true
    ]);
}
```

Côté client on vérifie que la réponse json est bien un succès et on redirige le client vers la page du panier, ce qui revient à actualiser la page actuelle, ce qui permet à l'utilisateur de voir son panier vide et le flashes lui annonçant que le paiement a été validé.

```
 const { success } = await response.json();
 if (success) {
     window.location.href = '/cart';
 }
```

3. Les livrables

Le projet est disponible via le dépôt GitHub contenant le code source du projet :
https://github.com/fb-lb/CEF_devoirs_stubborn

Le site web est hébergé à l'adresse suivante :
https://hyle.alwaysdata.net/

Il est possible d'y créer son propre compte, de s'y connecter avec un compte client déjà présent ou un compte admin déjà présent :

- Compte client :
  - User name : JohnDoe
  - Password : 1234567
- Compte admin :
  - User name : Admin
  - Password : 1234567

Pour tester le paiement il est possible d'utiliser les cartes proposées par Stripe :

- Paiement réussi : 4242424242424242
- Authentification requise pour le paiement : 4000002500003155
- Paiement refusé : 4000000000009995