

VI Fonctionnalité phare

1. Les services
2. La commande
3. Le panier
4. Le paiement

VI Fonctionnalité phare

1. Les Services

Les **services** sont des objets réutilisables qui encapsulent une logique métier spécifique (mise en session, gestion du panier, envoi d'email). Ils favorisent un code modulaire, maintenable et centralisé.

➤ **CartService** : Il est le cœur de la gestion du panier d'achat. Il centralise la logique d'ajout, de suppression et de modification des articles. Il assure également le stockage et la récupération du panier, et calcule les totaux.

Voici les **méthodes** utilisées pour la fonctionnalité phare :

Le **constructeur** : Il utilise **l'injection de dépendances** pour recevoir et stocker des **instances** de *RequestStack* (pour gérer la session), *TicketRepository* (pour accéder aux données des tickets) et *ExhibitionShareRepository* (pour accéder aux données des expositions). Cela permet au service d'accéder à ces composants sans avoir à les instancier lui-même.

```
public function __construct(  
    RequestStack $requestStack,  
    TicketRepository $ticketRepo,  
    ExhibitionShareRepository $exhibitionShareRepo)  
{  
    $this->ticketRepository = $ticketRepo;  
    $this->requestStack = $requestStack;  
    $this->exhibitionShareRepository = $exhibitionShareRepo;  
}
```

Méthode **getSession()** : Elle récupère l'objet *SessionInterface* via *RequestStack*. Déclarée *private*, elle encapsule l'accès à la session, contrôlant ainsi son utilisation et masquant la complexité de sa récupération. Cette encapsulation renforce la modularité du code : seul *CartService* interagit directement avec la gestion de la session. En limitant l'accès à cette méthode, on applique le principe de moindre privilège (droits d'accès minimaux), améliorant la sécurité et la robustesse.

```
private function getSession()
{
    return $this->requestStack->getCurrentRequest()->getSession();
}
```

Méthode **getCart()** : Elle récupère le contenu du panier de l'utilisateur à partir de la **session**. Elle extrait la valeur associée à la clé *\$cart* sous forme de tableau. Si le panier n'existe pas encore en session, elle retourne un tableau vide, ce qui évite les erreurs lorsqu'on essaie de manipuler un panier qui n'a pas encore été initialisé. Cela facilite la manipulation du panier dans d'autres parties de l'application.

```
public function getCart(): array
{
    return $this->getSession()->get('cart', []);
}
```

Méthode **setCart(array \$cart)** : Elle met à jour le tableau *\$cart* en session sous la clé 'cart' à chaque action de l'utilisateur qui affecte le contenu de son panier (ajout, suppression, modification de quantité). Cela permet de persister ainsi le contenu du panier de l'utilisateur à travers ses requêtes. Le terme "set" indique l'assignation du tableau *\$cart* à la clé 'cart' dans la session.

```
public function setCart(array $cart): void
{
    $this->getSession()->set('cart', $cart);
}
```

Méthode **addCart()** : Elle permet d'ajouter un ticket au panier en cliquant sur le plus.

```

/***** Ajouter un produit au panier *****/
public function addCart(Ticket $ticket, int $exhibitionId, int $qty = 1)
{
```

On récupère le panier actuel depuis la session. `$cart = $this->getCart();`

Mais aussi les détails du ticket via le *TicketRepository*, et l'exposition via l'*ExhibitionShareRepository*.

```
if ($ticket) {  
    // Récupérer les informations via le repository  
    $ticketId = $ticket->getId();  
    $ticketDetails = $this->ticketRepository->findTicketDetails($ticketId);  
}
```

On utilise une clé unique (`$exhibitionId.'_'.$ticketId`) pour identifier l'article dans le panier.

```
// Créer une clé unique combinant exhibitionId et ticketId  
$cartKey = $exhibitionId.'_'.$ticketId;
```

Si l'article existe déjà, on incrémente la quantité. Sinon, on ajoute un nouvel élément au panier avec les informations du ticket, de l'exposition, la quantité et le prix.

```
// Si le ticket est déjà dans le panier pour cette exposition, on met à jour la quantité  
if (isset($cart[$cartKey])) {  
    $cart[$cartKey]['qty'] += $qty;  
} else {  
    // Sinon, on ajoute le ticket au panier avec ses informations  
    $cart[$cartKey] = [  
        'ticket' => $ticket,  
        'ticketId' => $ticketId,  
        'exhibition' => $exhibition,  
        'exhibitionId' => $exhibitionId,  
        'qty' => $qty,  
        'price' => $price,  
    ];  
}
```

On calcule également le total de la ligne pour cet article (*totalLine*).

```
// Ajouter le total de la ligne  
$cart[$cartKey]['totalLine'] = $cart[$cartKey]['price'] * $cart[$cartKey]['qty'];
```

Puis on met à jour le panier dans la session et recalcule le total du panier.

```
// Sauvegarde du panier
$this->setCart($cart);
$this->updateCartTotal($cart);
```

Méthode **removeCart()** : Elle supprime 1 ticket en cliquant sur le moins.

```
/****** Soustraire un produit *****/
public function removeCart(Ticket $ticket, int $exhibitionId, int $qty = 1)
{
```

Récupère le panier et la clé de l'article par le biais de la clé unique lié au panier.

```
$cart = $this->getCart();
$ticketId = $ticket->getId();
$cartKey = $exhibitionId.'_'.$ticketId;
```

Si l'article existe, décrémente la quantité.

```
// Vérif si le ticket est présent dans le panier
if (isset($cart[$cartKey])) {
    $cart[$cartKey]['qty'] -= $qty;
```

Si la quantité devient inférieure ou égale à zéro, l'article est supprimé du panier.

```
//Suppression totale du ticket <= 0
if ($cart[$cartKey]['qty'] <= 0) {
    unset($cart[$cartKey]);
}
```

Met à jour le panier dans la session.

```
$this->setCart($cart);
```

Méthode **clearCart()** : Elle permet de supprimer le panier complet.

```
public function clearCart(): void
{
    $this->getSession()->remove('cart');
}
```

- **EmailService** : Il centralise l'envoi d'e-mails dans Symfony. Il utilise l'injection de dépendances pour utiliser *MailerInterface* pour l'envoi et Twig pour la mise en forme du contenu.

Voici les **méthodes** utilisées pour la fonctionnalité phare :

Méthode **send()** : Elle crée un objet *Email*, le configure (expéditeur, destinataire, sujet, corps HTML, pièce jointe) et l'envoie via *MailerInterface*. La méthode *renderTemplate()*

```
public function send(
    string $to, string $subject, string $body, ?array $attachment = null,
    string $from = 'noreply@regardsguerre.fr'
): void
{
    $email = (new Email())
        ->from($from)
        ->to($to)
        ->subject($subject)
        ->html($body);

    //Si le mail a des PJ alors on rajoute au mail
    if (is_array($attachment) && isset($attachment['content'], $attachment['filename'])) {
        $email->attach($attachment['content'], $attachment['filename'], $attachment
            ['mimeType'] ?? null);
    }

    $this->mailer->send($email);
}
```

Méthode **renderTemplate()** : Elle utilise *Twig* pour générer le contenu HTML de l'e-mail à partir d'un template et de variables. Ces méthodes encapsulent la logique d'envoi et de mise en forme des e-mails.

```
public function renderTemplate(string $templatePath, array $context = []): string
{
    return $this->twig->render($templatePath, $context);
}
```

- **StockAlertEmailService** : Il permet d'envoyer un mail à l'administrateur dès qu'un stock est presque épuisé ou épuisé.

```
public function sendStockAlertEmail(array $soonOutStockExhibits, array
$outOfStockExhibitions): void
{
    $body = $this->emailService->renderTemplate('emails/stockAlertEmail.html.
twig', [
        'soonOutStockExhibits' => $soonOutStockExhibits,
        'outOfStockExhibitions' => $outOfStockExhibitions,
    ]);

    $this->emailService->send('admin@regardsguerre.fr', 'Alerte de stock',
    $body);
}
```

- **OrderConfirmationEmailService** : Il se charge de générer le PDF de la commande et de l'envoyer à l'utilisateur.

Voici les **méthodes** utilisées pour la fonctionnalité phare :

Méthode **generateEticketPdf()** :

```
public function generateEticketPdf(Order $order): ?array
{
    //// Init du tabl pour les commandes regroupées
    $groupedOrderDetails = [];

    // Parcours des détails de la commande pour les regrouper par exposition
    foreach ($order->getOrderDetails() as $orderDetail) {
        $exhibitionId = $orderDetail->getExhibition()->getId();

        if (!isset($groupedOrderDetails[$exhibitionId])) {
            $groupedOrderDetails[$exhibitionId] = [
                'exhibition' => $orderDetail->getExhibition(),
                'orderDetails' => [],
            ];
        }
        // Ajoute le ticket à la liste
        $groupedOrderDetails[$exhibitionId]['orderDetails'][] = $orderDetail;
    }

    // Génère le contenu PDF (pdf service + template)
    $pdfContent = $this->pdfService->generatePdf(
        'pdf/eticketPdf.html.twig',
        [
            'order' => $order,
            'groupedOrderDetails' => $groupedOrderDetails,
        ]
    );

    // Retourne le contenu et le nom du fichier PDF
    return [
        'content' => $pdfContent,
        'filename' => 'eticket.pdf',
    ];
}
```

Méthode **sendTicketEmail()** :

```
public function sendTicketEmail(Order $order): void
{
    //Génère le pdf de la facture
    $orderPdf = $this->generateEticketPdf($order);

    if ($orderPdf) {
        $attachment = [
            'content' => $orderPdf['content'],
            'filename' => $orderPdf['filename'],
            'mimeType' => 'application/pdf',
        ];

        //Envoi l'email
        $this->emailService->send(
            $order->getCustomerEmail(),
            'Vos tickets',
            $this->emailService->renderTemplate(
                'emails/eticketEmail.html.twig',
                ['order' => $order]
            ),
            $attachment
        );
    }
}
```

➤ **OrderExportPdfService :**

Méthode **generateTicketsPdf()** : Elle génère les documents PDF (e-tickets) qui seront envoyés par e-mail. Elle récupère les informations de la commande et du panier, puis utilise un service PDF et un template Twig pour mettre en page chaque e-ticket. Enfin, elle retourne un tableau contenant le contenu PDF et le nom de fichier de chaque e-ticket.


```

public function generateTicketsPdf(int $orderId, array $groupedCart): ?array
{
    // Récupère l'entité Order à partir de son ID
    $order = $this->orderRepository->find($orderId);

    // Vérifie si la commande existe et est associée à un utilisateur
    if (!$order || !$order->getUser()) {
        return null; // Retourne null si la commande n'est pas trouvée
    }

    $tickets = [];

    // Vérifie si le panier regroupé contient des articles
    if (isset($groupedCart['items'])) {
        // Parcourt les articles du panier regroupé
        foreach ($groupedCart['items'] as $item) {
            // Vérifie si l'article est un ticket et est associé à une exposition
            if ($item['type'] === 'ticket' && isset($item['exposition'])) {
                // Génère le contenu PDF (pdf service + template)
                $pdfContentTicket = $this->pdfService->generatePdf(
                    'pdf/eticketPdf.html.twig',
                    ['item' => $item, 'order' => $order]
                );

                // Add le contenu et le nom du fichier de l'e-ticket au tableau
                $tickets[] = [
                    'content' => $pdfContentTicket,
                    'filename' => 'e_ticket_' . $item['exposition']['slug'] . '_' . $orderId . '.pdf',
                ];
            }
        }
    }

    // Retourne le tableau des e-tickets
    return $tickets;
}

```

➤ PdfService :

Méthode **generatePdf()** : Elle génère un PDF à partir d'un *template Twig*. Elle configure *Dompdf*, une bibliothèque PHP de conversion HTML en PDF, charge le HTML généré par Twig, définit le format du papier et retourne le contenu du PDF.


```

public function generatePdf(
    string $templatePath,
    array $data): string
{
    // Configuration des options de Dompdf
    $pdfOptions = new Options();
    $pdfOptions->set('defaultFont', 'Arial');
    $pdfOptions->set('isRemoteEnabled', true); // Pour placer le logo

    $dompdf = new Dompdf($pdfOptions); // Création d'une instance de Dompdf avec les options

    // Génération du HTML à partir du template Twig
    $html = $this->twig->render(
        $templatePath,
        $data
    );

    //Mep options dompdf
    $dompdf->loadHtml($html); // Chargement du HTML dans Dompdf
    $dompdf->setPaper('A4', 'portrait'); // Config du format de papier et de l'orientation
    $dompdf->render(); // Génération du PDF

    return $dompdf->output(); // Retourne le contenu du PDF
}

```

2. La commande

L'utilisateur se rend sur la fiche de l'exposition pour effectuer sa **commande**.

Ce template présente la vue détaillée d'une entité *Exhibition*, offrant une interface pour la consultation et l'interaction avec les *TicketPricings* associés.

Tickets :				
Adulte	10.00 €	-	2	+
Enfant	8.00 €	-	3	+
Enfant -6ans	Gratuit	-	0	+
Panier				

La section des tickets itère (*{% for pricing in ticketPricings %}*) sur les différents types, affichant leur *standardPrice*. L'interaction se fait via des liens générant des requêtes *GET* vers les routes *removeTicketFromCart* et *addTicketToCart*.

Le paramètre *origin* dans l'URL permet un traitement contextuel par le contrôleur (*CartController*). La quantité dans le panier est rendue dynamiquement via la variable *\$cart*, un tableau associatif basé sur une clé unique. Un bouton avec un lien hypertexte vers la route *cart* permet la navigation vers le récapitulatif.

```

{% if ticketPricings is not empty %}

    {% for pricing in ticketPricings %}
        <div class="ticketExhibit">
            <p class="titleTicket">{{ pricing.ticket.titleTicket }}</p>

            <div class="blockTitleQtyTicket">
                <div class="blockPriceTicket">
                    {% if pricing.standardPrice > 0 %}
                        <p class="price">{{ pricing.standardPrice }} €</p>
                    {% else %}
                        <p class="price">Gratuit</p>
                    {% endif %}
                </div>

                <div class="blockQtyTicket">

                    <a href="{{ path('removeTicketFromCart', {'exhibitionId': exhibition.id, 'ticketId': pricing.ticket.id, 'origin': 'listExhibit'}) }}"
                        aria-label="{{ 'Retirer un ticket ' ~ pricing.ticket.titleTicket |
                        lower ~ ' pour l\'exposition ' ~ exhibition.titleExhibit }}"
                        title="{{ 'Diminuer la quantité du ticket ' ~ pricing.ticket.titleTicket ~ ' pour l\'exposition ' ~ exhibition.titleExhibit }}">
                        <i class="fa-solid fa-minus"></i>
                    </a>

                    <div class="qtyProduct">
                        {# Crée une clé unique pour le ticket dans le panier (id expo + id ticket) #}
                        {% set cartKey = exhibition.id ~ '_' ~ pricing.ticket.id %}

                        {# Vérifie si le ticket est déjà dans le panier #}
                        {% if cart[cartKey] is defined %}
                            <span class="nbCart">{{ cart[cartKey].qty }}</span>
                        {% else %}
                            <span class="nbCart">0</span>
                        {% endif %}
                    </div>

                    <a href="{{ path('addTicketToCart', {'exhibitionId': exhibition.id, 'ticketId': pricing.ticket.id, 'origin': 'listExhibit'}) }}"
                        aria-label="{{ 'Ajouter un ticket ' ~ pricing.ticket.titleTicket |
                        lower ~ ' pour l\'exposition ' ~ exhibition.titleExhibit }}"
                        title="{{ 'Augmenter la quantité du ticket ' ~ pricing.ticket.titleTicket ~ ' pour l\'exposition ' ~ exhibition.titleExhibit }}">
                        <i class="fa-solid fa-plus"></i>
                    </a>
                </div>
            </div>
        </div>
    {% endfor %}
{% endif %}

```

3. Le panier

- **CartController** : Il est responsable de la gestion du panier d'achat de l'utilisateur. Il utilise le service *CartService* pour effectuer les opérations liées au panier (afficher, ajouter, supprimer, vider, calculer le total, regrouper les articles).

Le constructeur : Il injecte une instance du service *CartService*, rendant ses méthodes accessibles dans le contrôleur via la propriété `$this->cartService`.

```
public function __construct(
    CartService $cartService)
{
    $this->cartService = $cartService;
}
```

Méthode **showCart()** : Elle affiche le contenu du panier.

```
#[Route('/order/cart', name: 'cart')]
public function showCart(?User $user): Response
```

Elle récupère le panier, le total et le panier regroupé par exposition en utilisant les méthodes correspondantes du *CartService*.

```
// Récupérer le panier depuis le service
$cart = $this->cartService->getCart();
$total = $this->cartService->getTotal();
$groupedCart = $this->cartService->groupCartByExhibition($cart);
```

Elle crée également une instance du formulaire *UserIdentityCartFormType*, qui est utilisé pour recueillir les informations d'identité de l'utilisateur avant la commande (si elles ne sont pas déjà renseignées).

```
//Obl pour commander
$form = $this->createForm(UserIdentityCartFormType::class);
```

Enfin, elle rend le *template order/cart.html.twig* en passant les informations du panier, le total et le formulaire. Le paramètre `?User $user` permet d'accéder à l'utilisateur connecté.

```

return $this->render('order/cart.html.twig', [
    'groupedCart' => $groupedCart,
    'cart' => $cart,
    'total' => $total,
    'form' => $form->createView(),
]);

```

Méthode **addTicketToCart()** : Elle permet d'ajouter un ticket spécifique au panier. Elle reçoit l'ID de l'exposition (*exhibitionId*) et l'ID du ticket (*ticketId*) via les paramètres de la route, ainsi que l'origine de la requête (*origin*).

```

/***** Ajoute un ticket au panier *****/
#[Route('/ticket/{exhibitionId}/addTicketToCart/{ticketId}/{origin}', name: 'addTicketToCart')]
public function addTicketToCart(

```

Elle utilise le *TicketRepository* pour récupérer l'entité Ticket correspondant à l'*\$ticketId* et l'*ExhibitionShareRepository* pour récupérer l'entité *Exhibition* correspondant à l'*\$exhibitionId*.

```

TicketRepository $ticketRepo,
int $exhibitionId,
int $ticketId,
string $origin,
ExhibitionShareRepository $exhibitionShareRepo): Response
{
    // Récup du ticket via le repository
    $ticket = $ticketRepo->find($ticketId);

    // Récup de l'exposition via le repository en utilisant l'ID de la route
    $exhibition = $exhibitionShareRepo->find($exhibitionId);

```

Elle appelle la méthode *addCart()* du *CartService* pour ajouter le ticket à la session du panier, en fournissant l'entité Ticket et l'*\$exhibitionId*.

```

// Ajout du ticket au panier via le service
$this->cartService->addCart($ticket, $exhibitionId);

```

Elle effectue une redirection en fonction de la valeur du paramètre *\$origin*. Si *\$origin* est 'listExhibit', l'utilisateur est redirigé vers la page de l'exposition. Sinon, il est redirigé vers la page du panier.

```
// Redirection en fonction de l'origine
if ($origin === 'listExhibit') {
    return $this->redirectToRoute('exhibition', ['slug' => $exhibition->getSlug()]);
}

// Redirection par défaut
return $this->redirectToRoute('cart', ['exhibition' => $exhibitionId]);
```

Méthode **removeTicketFromCart()** : Elle supprime un ticket du panier. Elle reçoit les mêmes paramètres que *addTicketToCart*.

```
/****** Soustrait un produit au panier *****/
#[Route('/ticket/{exhibitionId}/removeTicketFromCart/{ticketId}/{origin}', name:
'removeTicketFromCart')]
public function removeTicketFromCart(
    TicketRepository $ticketRepo,
    int $exhibitionId,
    int $ticketId,
    string $origin,
    ExhibitionShareRepository $exhibitionRepo) : Response
{
    // Récupération du ticket via le repository
    $ticket = $ticketRepo->find($ticketId);

    // Récup de l'exposition via le repository en utilisant l'ID de la route
    $exhibition = $exhibitionRepo->find($exhibitionId);
```

Elle appelle la méthode *removeCart()* du *CartService* pour diminuer la quantité du ticket dans le panier pour l'exposition donnée.

```
// Soustraction au panier via le service
$this->cartService->removeCart($ticket, $exhibitionId);
```

Elle effectue une redirection vers la page de l'exposition si l'origine est 'listExhibit', ou vers la page du panier si l'origine est 'cart'.

```
if ($origin === 'listExhibit') { // Si l'origine est 'listExhibit', rediriger vers la page de
ventes des tickets
    return $this->redirectToRoute('exhibition', ['slug' => $exhibition->getSlug()]);
}

if ($origin === 'cart') { // Si l'origine est 'cart', rediriger vers le panier
    return $this->redirectToRoute('cart', [
        'exhibition' => $exhibitionId,
    ]);
}

// Par défaut : rediriger vers le panier
return $this->redirectToRoute('cart');
```


Méthode **getTotal()** : Calcule le total du panier en parcourant tous les articles et en multipliant leur prix par leur quantité.

```
/****** Calcul total du panier *****/
public function getTotal(): float
{
    $cart = $this->getCart(); // Récupération du panier depuis la session
    $total = 0;

    foreach ($cart as $product) {
        $total += $product['price'] * $product['qty'];
    }

    return $total;
}
```

Méthode **updateCartTotal()** : Recalcule le total du panier en se basant sur le *\$totalLine* de chaque article et met à jour la valeur *'cartTotal'* dans la session.

```
// Fonction pour mettre à jour le total du panier
public function updateCartTotal($cart)
{
    $total = 0;

    foreach ($cart as $product) {
        $total += $product['totalLine']; // Total de la ligne du ticket
    }

    // Mettre à jour le total dans la session
    $session = $this->requestStack->getCurrentRequest()->getSession();
    $session->set('cartTotal', $total);
}
```

Méthode **groupCartByExhibition()** : Elle regroupe les articles du panier par exposition.

```
/****** Regroupement des achats *****/
public function groupCartByExhibition(array $cart): array
{
    // ...
}
```

On initialise un tableau. On boucle sur les expositions du panier. Si l'exposition n'existe pas on l'ajoute au tableau avec toutes ses informations et on initialise un tableau de tickets, sinon on passe à la suivante.

```
// Init tableau de regroupement
$groupedCart = [];

foreach ($cart as $product) {
    // Récupération des IDs pour plus de lisibilité
    $exhibitionId = $product['exhibitionId'];
    $ticketId = $product['ticketId'];

    // Si expo n'existe pas on la crée
    if (!isset($groupedCart[$exhibitionId])) {
        $groupedCart[$exhibitionId] = [
            'exhibition' => $product['exhibition'], // Add infos expo
            'tickets' => [] // Init tabl tickets
        ];
    }
}
```

Maintenant on vérifie si le ticket n'existe pas on le crée avec toutes les informations (quantité, prix, nom) sinon on le crée.

```
// Si ticket n'existe pas on le crée
if (!isset($groupedCart[$exhibitionId]['tickets'][$ticketId])) {
    $groupedCart[$exhibitionId]['tickets'][$ticketId] = [
        'ticket' => $product['ticket'], // Add infos ticket
        'quantity' => $product['qty'],
        'price' => $product['price'],
        // Ajout optionnel du total ligne si nécessaire
        'totalLine' => $product['price'] * $product['qty']
    ];
} else {
    // Si ticket existe on incrémente qty
    $groupedCart[$exhibitionId]['tickets'][$ticketId]['quantity'] += $product['qty'];
    // Mise à jour du total ligne si vous l'avez ajouté
    if (isset($groupedCart[$exhibitionId]['tickets'][$ticketId]['totalLine'])) {
        $groupedCart[$exhibitionId]['tickets'][$ticketId]['totalLine'] += $product['price'] *
        $product['qty'];
    }
}
```

4. Le paiement

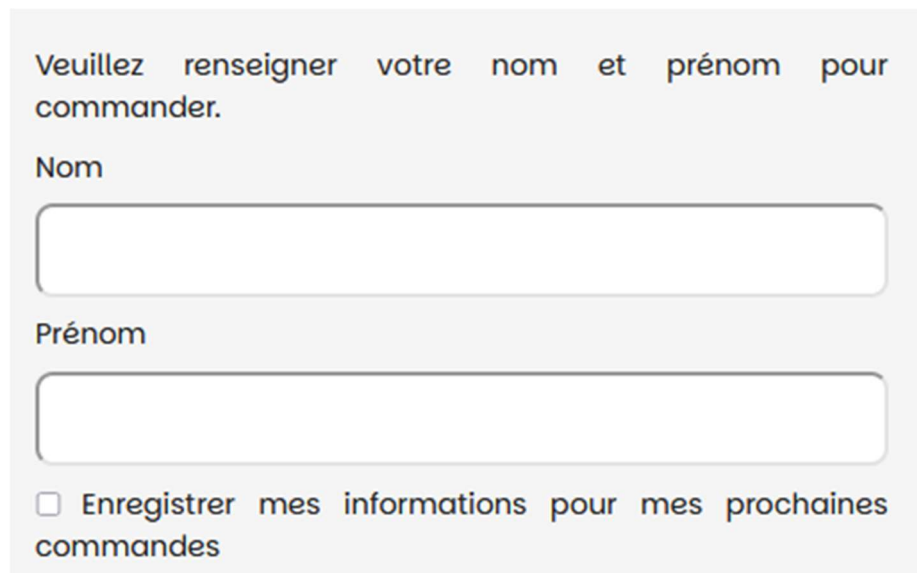
Une fois que l'utilisateur a constitué ses achats, il se rend dans son panier.

Si les champs *userName* et *userFirstname* sont déjà renseignés dans son entité *User*, il visualise directement le récapitulatif de sa commande ainsi que le bouton de paiement.

Si son identité (nom et prénom) n'est pas renseignée dans son profil, un formulaire sera présent. Il est obligatoire de renseigner son identité pour

commander, notamment en raison des obligations légales liées à la vente de tickets nominatifs et à la facturation. En effet, la vente de tickets nominatifs requiert l'identification de l'acheteur pour des raisons de conformité et de traçabilité.

L'utilisateur a également la possibilité de choisir de persister ou non ces données via la case à cocher *saveIdentity*, conformément au *RGPD* et au principe de *minimisation des données*.



Veillez renseigner votre nom et prénom pour commander.

Nom

Prénom

☐ Enregistrer mes informations pour mes prochaines commandes

Lors de la soumission de ce formulaire (*méthode HTTP POST*), les données sont transmises au contrôleur.

➤ **PaymentController**

Stripe est l'API qui a été utilisée. On envoie une *requête HTTP* à l'*API de Stripe*, avec des données (détails de paiement). *Stripe* traite la requête, effectue les opérations (débit d'une carte), et renvoie une *réponse HTTP*. La réponse contient des données structurées indiquant le résultat (réussite/échec). L'*API de Stripe* est sécurisée : elle utilise des **clés d'authentification** et **chiffre** les données sensibles. Ce processus permet d'intégrer des paiements complexes sans gérer les détails techniques et de sécurité.

Voici les **méthodes** utilisées pour la fonctionnalité phare :

Méthode **stripeCheckout()** :

```
#[Route('/order/create-session-stripe', name: 'paymentStripe')]
public function stripeCheckout()
```

Ce contrôleur crée une instance du *UserIdentityCartFormType*.

```
$form = $this->createForm(UserIdentityCartFormType::class, $this->getUser());
```

Si les champs ont été remplis alors on réceptionne la requête.

```
//Si nom prenom !bdd
if (!$this->getUser()->getUserName() && !$this->getUser()->getUserFirstname()) {
    $form->handleRequest($request);
}
```

Ensuite si les données saisies sont valides à l'issue de la soumission du formulaire alors on sauvegarde les informations en session.

```
if ($form->isSubmitted() && $form->isValid()) {
    //Save en session car paiement non validé
    $session = $this->requestStack->getCurrentRequest()->getSession();
    $session->set('customerName', $form->get('userName')->getData());
    $session->set('customerFirstname', $form->get('userFirstname')->getData());
    $session->set('saveIdentity', $form->get('saveIdentity')->getData());
}
```

Sinon on notifie l'utilisateur d'un problème par le biais d'un message flash et on lui réaffiche son panier (via le *CartService*) ainsi que le formulaire.

```
} else {
    // Ajouter un message flash d'erreur
    $this->addFlash(
        'error',
        'Veuillez renseigner votre nom et prénom pour continuer la commande.'
    );

    // Réafficher le form avec les erreurs
    return $this->render('order/cart.html.twig', [
        'form' => $form->createView(), //Converti l'objet form en format Twig
        'groupedCart' => $this->cartService->groupCartByExhibition($this->cartService->getCart()),
        'cart' => $this->cartService->getCart(),
        'total' => $this->cartService->getTotal(),
    ]);
}
```

Une vérification du stock disponible est effectuée pour cela on récupère le panier.

```
//Récup panier pour vérif stock + préparation donnée pour Stripe
$cart = $this->cartService->getCart();
```

On initialise un tableau pour y stocker les expositions dont le stock ne correspond pas à la quantité demandée. Puis on effectue les vérifications en comparant la quantité demandée avec le stock maximal moins les tickets déjà réservés pour l'exposition concernée.

```
// Vérification init du stock et collecte des erreurs de stock
foreach ($cart as $item) {
    $exhibition = $exhibitionShareRepo->find($item['exhibitionId']); //Récup id expo like id dans panier
    if ($exhibition) {
        //Vérif si les tickets commandés sont dispos
        $ticketsAvailable = $exhibition->getStockMax() - $exhibition->getTicketsReserved();
        $qtyRequested = $item['qty']; //Qté demandée

        if ($qtyRequested > $ticketsAvailable) {
            $stockErrors[] = [
                'exhibitionTitle' => $exhibition,
                'ticketsAvailable' => $ticketsAvailable,
            ];
        }
    }
}
```

Si un stock insuffisant est détecté pour un ou plusieurs articles, un message flash de type danger est affiché pour chaque erreur (nom de l'exposition ainsi que le nombre de ticket restant disponible), informant l'utilisateur du problème, et il est redirigé vers la page du panier pour modifier sa commande.

```
if (!empty($stockErrors)) {
    foreach ($stockErrors as $error) {
        $this->addFlash(
            'danger',
            sprintf( //Permet concaténation string + variable
                'Stock insuffisant pour l\'exposition "%s". (%d restants).',
                $error['exhibitionTitle'],
                $error['ticketsAvailable']
            )
        );
    }
}
```

Si le stock est suffisant pour tous les articles, un tableau contenant les informations nécessaires pour l'API Stripe (*\$productStripe*) est initialisé. Pour chaque article du panier, les informations du ticket (nom) et le prix unitaire sont récupérés, et un tableau structuré pour Stripe est créé, incluant la devise (EUR) et la quantité.

```

$productStripe = [];

foreach ($cart as $product) {
    $ticket = $ticketRepo->find($product['ticketId']);
    $exhibition = $exhibitionShareRepo->find($product['exhibitionId']);
    $qty = $product['qty'];
    $price = $product['price'];

    $productStripe[] = [
        'price_data' => [
            'currency' => 'eur',
            'product_data' => [
                'name' => $ticket->getTitleTicket(),
            ],
            'unit_amount' => $price * 100, // Prix en centimes
        ],
        'quantity' => $qty,
    ];
}

```

On configure la clé API de Stripe pour l'authentification (clé à sécuriser dans le fichier `.env` comme pour toutes les données sensibles)

```
Stripe::setApiKey($_ENV['STRIPE_SECRET_KEY']);
```

Et on génère les URLs de succès et d'échec de paiement.

```

$successUrl = $urlGenerator->generate('paymentSuccess', [], UrlGeneratorInterface::ABSOLUTE_URL);
$cancelUrl = $urlGenerator->generate('paymentError', [], UrlGeneratorInterface::ABSOLUTE_URL);

```

Une session de paiement Stripe est alors créée via l'API Stripe, incluant l'adresse e-mail de l'utilisateur (`$this->getUser()->getUserIdentifier()`), le mode de paiement (`payment_method_types` réglé sur `'card'`), les `line_items` représentant les produits du panier (basés sur le tableau `$productStripe`) et les URLs de succès (`success_url` générée pour la route `paymentSuccess`) et d'annulation (`cancel_url` générée pour la route `paymentError`).

```

$checkoutSession = Session::create([
    'customer_email' => $this->getUser()->getUserIdentifier(),
    'payment_method_types' => ['card'],
    'line_items' => $productStripe,
    'mode' => 'payment',
    'success_url' => $successUrl,
    'cancel_url' => $cancelUrl,
]);

```

Enfin, l'utilisateur est redirigé vers l'URL de paiement fournie par Stripe.

```

return new RedirectResponse($checkoutSession->url, 303);

```

En cas d'échec du paiement (retour via *cancel_url* et la route *paymentError*), l'utilisateur est redirigé vers la page du panier avec un message flash de type danger indiquant une erreur de paiement.

```

#[Route('/order/error', name: 'paymentError')]
public function stripeError(
    CartService $cartService,
): Response
{
    $this->addFlash('danger', 'Une erreur est survenue lors du paiement. Veuillez réessayer.');
```

```

    return $this->redirectToRoute('cart');
}

```

En cas de succès du paiement (retour via *success_url* et la route *paymentSuccess*), les informations du formulaire stockées en session sont récupérées.

```

//Récup des infos du form stockés en session
$session = $requestStack->getCurrentRequest()->getSession();
$customerName = $session->get('customerName');
$customerFirstname = $session->get('customerFirstname');
$saveIdentity = $session->get('saveIdentity');

```

Si l'utilisateur a activé l'option d'enregistrement (*saveIdentity* est *true*), les propriétés *userName* et *userFirstname* de l'entité *User* sont mises à jour avec les données de session, puis *l'EntityManager Doctrine* est utilisé pour persister et enregistrer ces modifications en base de données. Un message flash de type *success* informe l'utilisateur de cet enregistrement.


```

$user = $this->getUser(); //Récup user co

//Si $saveIdentity = true alors enregistrement dans user
if ($saveIdentity && $user) {
    $user->setUserName($customerName);
    $user->setUserFirstname($customerFirstname);
    $this->entityManager->persist($user);
    $this->entityManager->flush();
    $this->addFlash('success', 'Vos informations ont été enregistrées pour vos prochaines commandes.');
```

Ensuite, le contenu du panier est récupéré via le *CartService*.

```

$cart = $this->cartService->getCart(); /
```

Une nouvelle instance de l'entité *Order* est créée et hydratée avec les informations pertinentes : *orderDateCreation* (avec un objet *\DateTimeImmutable*), l'entité *User* associée, *orderStatus* initialisé à 'Envoyé', le nom et prénom du client (priorité aux données de session, sinon celles de l'entité *User*), l'adresse e-mail de l'utilisateur et le *orderTotal* calculé par le *CartService*.

```

// Vérif si un user est co ET si son nom et prénom ne sont pas vides
$order = new Order();
$order->setOrderDateCreation(new \DateTimeImmutable()); // //Avec une date immuable (const)
$order->setUser($this->getUser()); // Associe la commande au user co
$order->setOrderStatus('Envoyé');

// Récup le nom et prénom du client. Priorité à la session (formulaire), sinon BDD de l'utilisateur connecté.
$order->setCustomerName($customerName);
$order->setCustomerFirstname($customerFirstname);

$order->setCustomerEmail($this->getUser()->getUserIdentifier()); // Enregistre l'email de l'utilisateur

// Enregistrement du total de la commande
$total = $cartService->getTotal(); // Récupère le total du panier
$order->setOrderTotal($total); // Enregistre le total dans la commande
```

L'*EntityManager Doctrine* prépare et exécute l'enregistrement de cette entité *Order* en base de données. On itère ensuite sur les éléments du panier pour créer une nouvelle instance de l'entité *OrderDetail* pour chaque article. Chaque *OrderDetail* est hydraté avec l'entité *Order* associée, l'entité *ExhibitionShare* et l'entité *Ticket* correspondantes (retrouvées via leurs repositories), la *quantity* et le *unitPrice*.

```

// Création des détails de la commande
foreach ($cart as $item) {
    $orderDetail = new OrderDetail();
    $orderDetail->setOrder($order);

    // Charger l'objet Exhibition à partir de l'ID
    $exhibition = $exhibitShareRepo->find($item['exhibitionId']);
    if ($exhibition) {
        $orderDetail->setExhibition($exhibition);
    }

    // Charger l'objet Ticket à partir de l'ID
    $ticket = $ticketRepo->find($item['ticketId']);
    if ($ticket) {
        $orderDetail->setTicket($ticket);
    }

    $orderDetail->setQuantity($item['qty']);
    $orderDetail->setUnitPrice($item['price']);

    $this->entityManager->persist($orderDetail);
}

```

Ces entités *OrderDetail* sont ensuite persistées via *l'EntityManager*.

```

// Persist de l'order AVANT de générer le numéro de facture
$this->entityManager->persist($order);
$this->entityManager->flush();

```

L'entité *Order* est rafraîchie (*\$this->entityManager->refresh(\$order)*) afin de récupérer l'ID généré par la base de données, qui sera utilisé pour la création du numéro de facture unique.

```

// Rafraîchir l'entité pour récupérer l'ID généré par la bdd
$this->entityManager->refresh($order);

```

Une nouvelle instance de l'entité *Invoice* est instanciée et hydratée avec les informations de la commande : *customerName*, *customerFirstname*, *customerEmail*, *orderTotal* et *dateInvoice* (avec un objet *\DateTimeImmutable*).


```
// Création de la facture
$invoice = new Invoice();
$invoice->setCustomerName($order->getCustomerName());
$invoice->setCustomerFirstname($order->getCustomerFirstname());
$invoice->setCustomerEmail($order->getCustomerEmail());
$invoice->setOrderTotal($order->getOrderTotal());
$invoice->setDateInvoice(new \DateTimeImmutable()); // Utilise une date immutable
```

Un *numberInvoice* unique est généré en utilisant l'ID de la commande et la date de création de la commande (*orderDateCreation*). Ce numéro est ensuite attribué aux propriétés *numberInvoice* des entités *Order* et *Invoice*.

```
// Génération du numéro de facture unique
$orderId = $order->getId();
$orderDate = $order->getOrderDateCreation()->format('Ymd');

$invoiceNumber = sprintf('%s-%s', $orderDate, $orderId);
$order->setNumberInvoice($invoiceNumber);
$invoice->setNumberInvoice($invoiceNumber);
```

Un *slug* unique et lisible pour l'administrateur est créé en combinant l'ID de l'utilisateur, son nom et son prénom, et est attribué à la propriété *slug* de l'entité *Invoice*.

```
// Génération du slug avec l'ID, le nom et le prénom de l'utilisateur
$userId = $order->getUser()->getId(); // Récupère l'ID de l'utilisateur à partir de l'entité $order
$userName = $invoice->getCustomerName();
$userFirstname = $invoice->getCustomerFirstname();

$slug = sprintf('%d-%s-%s', $userId, $userName, $userFirstname);
$invoice->setSlug($slug);
```

On itère sur le panier pour extraire les détails de chaque article (titres de l'exposition et du ticket, prix, quantité) et les structurer dans un tableau *\$invoiceDetails*. Ce tableau, converti en **JSON** par **sérialisation**, est ensuite affecté à la propriété *invoiceDetails* de l'entité *Invoice* pour être stocké en base de données. (Lors de l'affichage de la facture, ces données JSON seront **désérialisées** pour retrouver la structure PHP originale.)

```

$invoiceDetails = []; //Détail de la commande

//Récup les éléments du panier
foreach ($cart as $item) {
    $exhibition = $exhibitShareRepo->find($item['exhibitionId']);
    $ticket = $ticketRepo->find($item['ticketId']);
    $quantity = $item['qty'];
    $price = $item['price'];

    $invoiceDetails[] = [
        'exhibitionTitle' => $exhibition->getTitleExhibit(),
        'ticketTitle' => $ticket->getTitleTicket(),
        'standardPrice' => $price,
        'quantity' => $quantity,
    ];
}

$invoice->setInvoiceDetails($invoiceDetails);

```

Ensuite, l'entité *Order* et l'entité *Invoice* sont persistées et enregistrées en base de données via *EntityManager*.

```

$this->entityManager->persist($order);
$this->entityManager->persist($invoice);
$this->entityManager->flush();

```

L'envoi de la confirmation de commande à l'utilisateur est géré par le service *OrderConfirmationEmailService* via sa méthode *sendTicketEmail()*. Ce service utilise le *EmailService* pour l'envoi d'e-mails et un service de génération de PDF pour les e-tickets.

```

$this->orderConfirmEmailService->sendTicketEmail($order);

```

L'envoi d'une alerte de stock à l'administrateur est géré par le service *StockAlertEmailService* via sa méthode *sendStockAlertEmail()*.

Avant cet envoi, on regroupe les articles du panier l'entité *Exhibition* est rafraîchie pour obtenir les dernières informations de stock.

```

$groupedCart = $this->cartService->groupCartByExhibition($cart);

$this->entityManager->refresh($exhibition); //rafraichit l'expo

```

Ensuite, on boucle sur le *\$groupedCart* (le panier regroupé par exposition) pour vérifier le stock restant de chaque exposition. Si le stock restant est inférieur ou égal au seuil d'alerte (*stockAlert*) et supérieur à zéro, l'exposition est ajoutée au tableau *\$soonOutStockExhibits*.

Si le stock est à zéro, l'exposition est ajoutée au tableau *\$outOfStockExhibitions*.

```
// Vérification des stocks APRÈS l'enregistrement de la commande en parcourant le groupedCart
foreach ($groupedCart as $exhibitionId => $items) { //Parcours groupedCart (tab1 AM)
    $exhibition = $exhibitShareRepo->find($exhibitionId);

    if ($exhibition) {
        $remainingStock = $exhibition->getStockMax() - $exhibition->getTicketsReserved(); // Calcul du stock
        restant

        if ($remainingStock <= $exhibition->getStockAlert() && $remainingStock > 0 && !in_array($exhibition,
        $soonOutStockExhibits)) {
            $soonOutStockExhibits[] = $exhibition; // si stock presque épuisé alors expo dans le tableau
        }
        if ($remainingStock <= 0 && !in_array($exhibition, $outOfStockExhibitions)) {
            $outOfStockExhibitions[] = $exhibition; // si stock épuisé alors expo dans le tableau
        }
    }
}
```

Le service *StockAlertEmailService* utilise ensuite le *EmailService* pour envoyer un e-mail à l'administrateur si l'un de ces tableaux n'est pas vide.

```
if (!empty($soonOutStockExhibits) || !empty($outOfStockExhibitions)) {
    $this->stockAlertEmailService->sendStockAlertEmail(array_unique($soonOutStockExhibits),
    array_unique($outOfStockExhibitions));
}
```

Enfin, le panier de l'utilisateur est vidé en utilisant la méthode *clearCart()* du service *CartService*. L'utilisateur est ensuite redirigé vers une page de succès de commande (route *orderSuccess*).

```
$this->cartService->clearCart();

return $this->redirectToRoute('orderSuccess');
```