VI Fonctionnalité phare

1. Les services

2. La commande

3. Le panier

4. Le paiement

VI Fonctionnalité phare

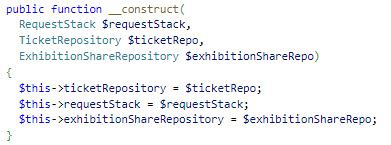
1. Les Services

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.

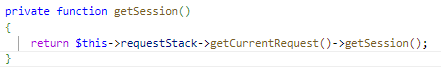
* **CartService** : Il est le cœur de la gestion du panier d'achat. Il centralise la logique d'ajout, de suppression et de modification des articles. Il assure également le stockage et la récupération du panier, et calcule les totaux.

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

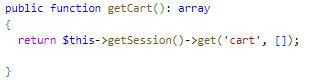
Le **constructeur** : Il utilise **l'injection de dépendances** pour recevoir et stocker des **instances** de *RequestStack* (pour gérer la session), *TicketRepository* (pour accéder aux données des tickets) et *ExhibitionShareRepository* (pour accéder aux données des expositions). Cela permet au service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



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.



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.



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.

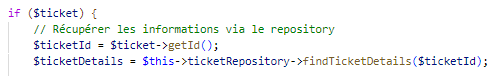


Méthode **addCart()** : Elle permet d’ajouter un ticket au panier en cliquant sur le plus.



On récupère le panier actuel depuis la session.

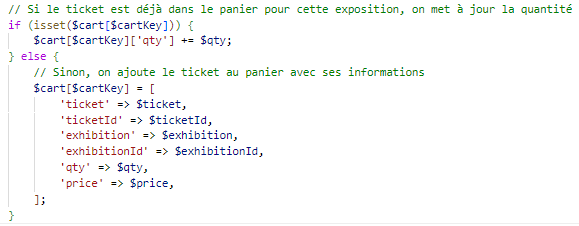
Mais aussi les détails du ticket via le *TicketRepository*, et l'exposition via l'*ExhibitionShareRepository*.



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



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.



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



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



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



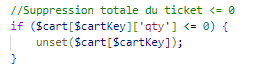
Récupère le panier et la clé de l'article par le biais de la clé unique lié au panier.



Si l'article existe, décrémente la quantité.



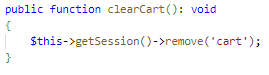
Si la quantité devient inférieure ou égale à zéro, l'article est supprimé du panier.



Met à jour le panier dans la session.



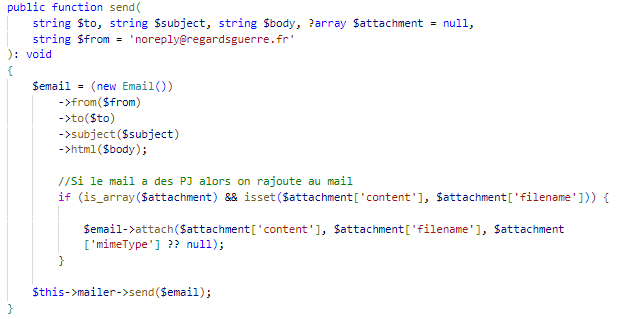
Méthode **clearCart()** : Elle permet de supprimer le panier complet.

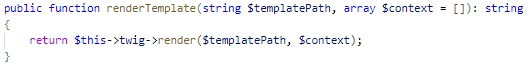


* **EmailService** : Il centralise l'envoi d'e-mails dans Symfony. Il utilise l'injection de dépendances pour utiliser *MailerInterface* pour l'envoi et Twig pour la mise en forme du contenu.

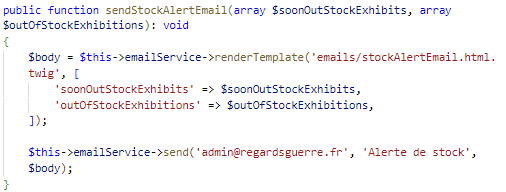
Voici les **méthodes** utilisées pour la fonctionnalité phare :

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



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.

* **StockAlertEmailService** : Il permet d’envoyer un mail a l’administrateur dès qu’un stock est presque épuisé ou épuisé.



* **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()** :

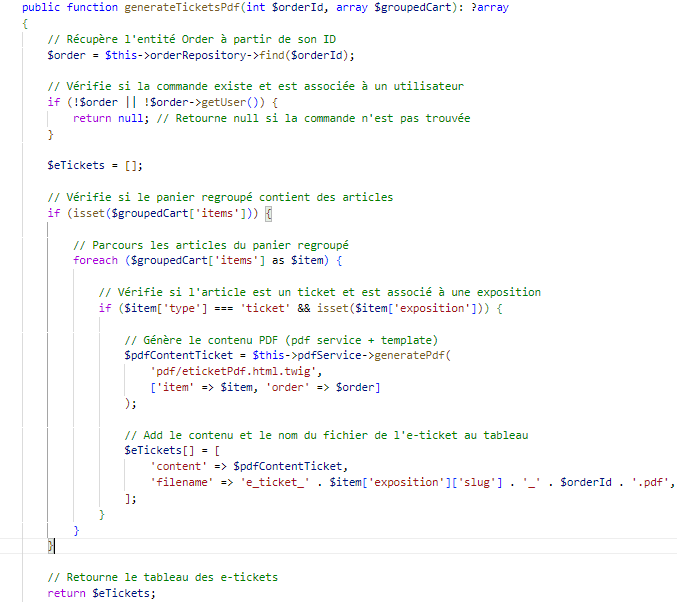




Méthode **sendTicketEmail()** :

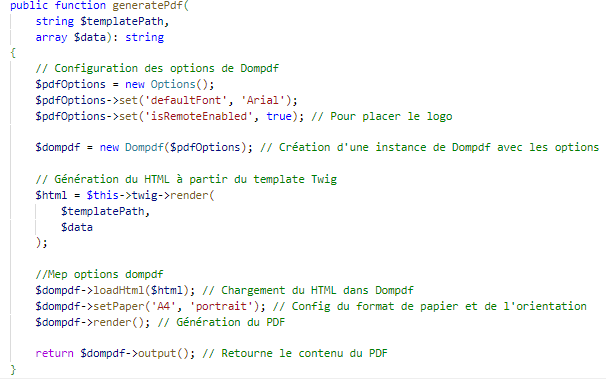
* **OrderExportPdfService** :

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.



* **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.



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.

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*.

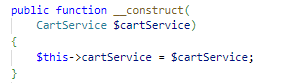
Le paramètre origin dans l'URL permet un traitement contextuel par le contrôleur (*CartController*). La quantité dans le panier est rendue dynamiquement via la variable $*cart*, un tableau associatif basé sur une clé unique. Un bouton avec un lien hypertexte vers la route cart permet la navigation vers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



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).

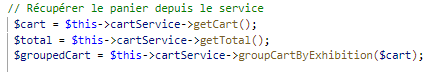
Le **contructeur** : Il injecte une instance du service *CartService*, rendant ses méthodes accessibles dans le contrôleur via la propriété $this->cartService.



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



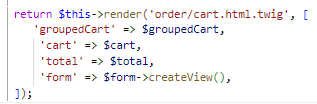
Elle récupère le panier, le total et le panier regroupé par exposition en utilisant les méthodes correspondantes du *CartService*.



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).



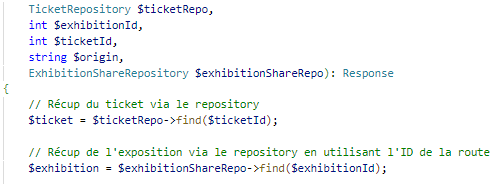
Enfin, elle rend le *template order/cart.html.twig* en passant les informations du panier, le total et le formulaire. Le paramètre *?User $user* permet d'accéder à l'utilisateur connecté.



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*).

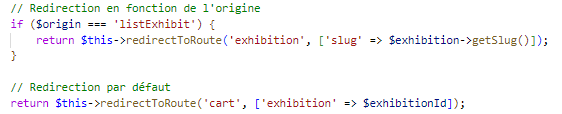


Elle utilise le *TicketRepository* pour récupérer l'entité Ticket correspondant à l'*$ticketId* et *l'ExhibitionShareRepository* pour récupérer l'entité *Exhibition*correspondant à l'*$exhibitionId*.

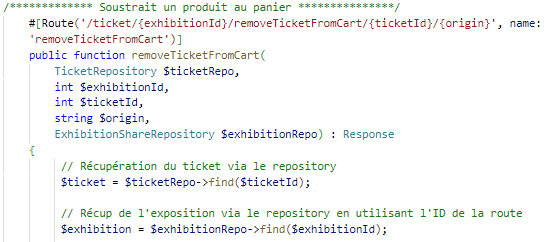


Elle appelle la méthode *addCart()* du *CartService* pour ajouter le ticket à la session du panier, en fournissant l'entité Ticket et l'*$exhibitionId*.



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.

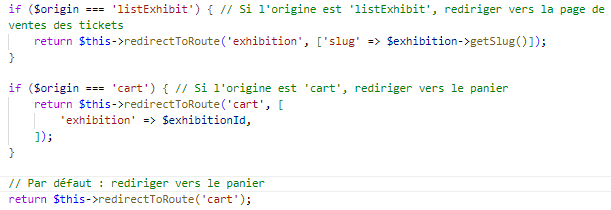
Méthode **removeTicketFromCart()** : Elle supprime un ticket du panier. Elle reçoit les mêmes paramètres que *addTicketToCart*.



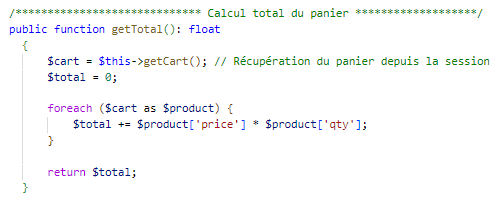
Elle appelle la méthode *removeCart()* du *CartService* pour diminuer la quantité du ticket dans le panier pour l'exposition donnée.



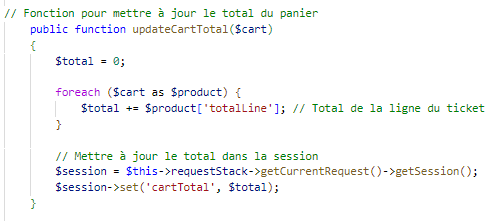
Elle effectue une redirection vers la page de l'exposition si l'origine est *'listExhibit'*, ou vers la page du panier si l'origine est *'cart'*.



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



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.

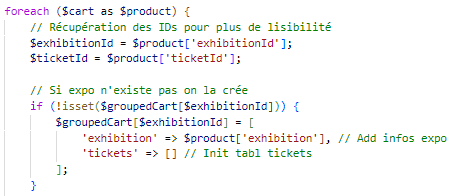


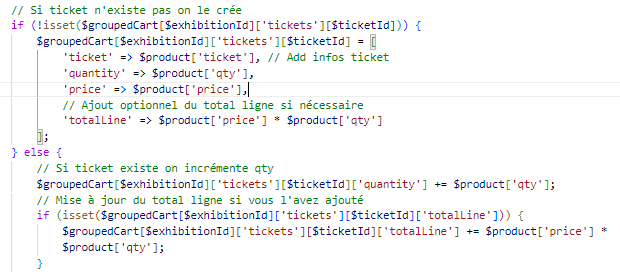
Méthode **groupCartByExhibition()** : Elle regroupe les articles du panier par exposition.



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.





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

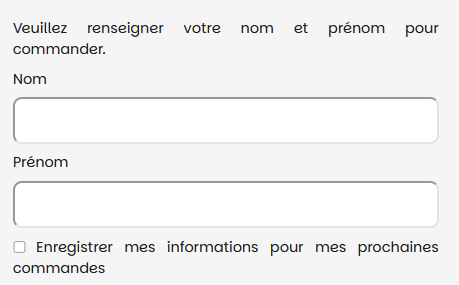
4. Le paiement

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*.



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

* **PaymentController**

**Stripe** est l’API qui a utilisée. On envoie une *requête HTTP* à *l'API de Stripe*, avec des données (détails de paiement). *Stripe* traite la requête, effectue les opérations (débiter 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 :

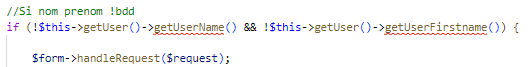
Méthode **stripeCheckout()** :



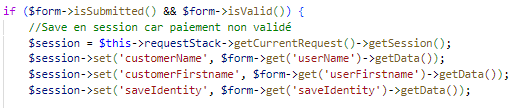
Ce contrôleur crée une instance du *UserIdentityCartFormType*.

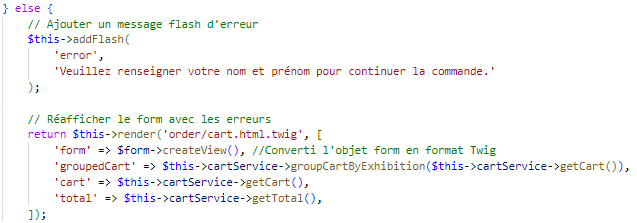


Si les champs ont été remplis alors on réceptionne la requête.



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

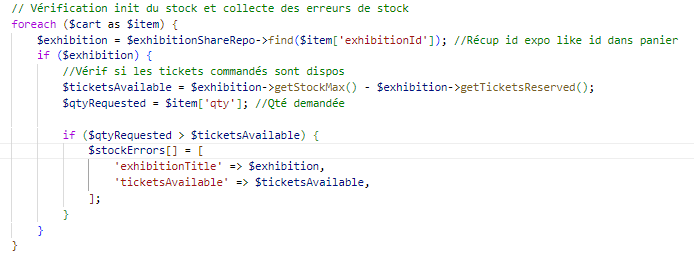


Sinon on notifie l’utilisateur d’un problème par le biais d’un message flash et on lui réaffiche son panier (via le *CartService*) ainsi que le formulaire.

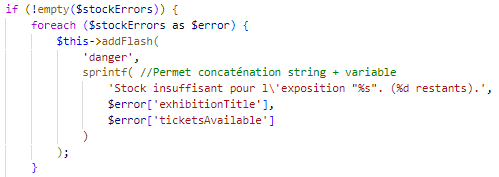
Une vérification du stock disponible est effectuée pour cela on récupère le panier.



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.



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.



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.



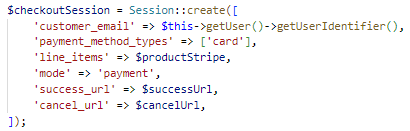
On configure la clé API de Stripe pour l’authentification (clé a sécuriser dans le fichier *‘.env’* comme pour toutes les données sensibles)



Et on génère les URLS de succès et d’échec de paiement.



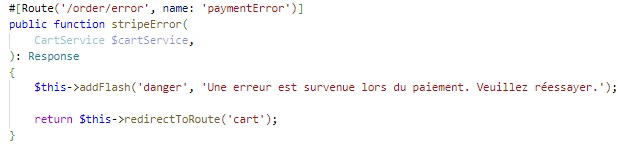
Une session de paiement Stripe est alors créée via l'API Stripe, incluant l'adresse e-mail de l'utilisateur (*$this->getUser()->getUserIdentifier()*), le mode de paiement (*payment\_method\_types* réglé sur *'card'*), les line\_items représentant les produits du panier (basés sur le tableau *$productStripe*) et les URLs de succès (*success\_url* générée pour la route *paymentSuccess*) et d'annulation (*cancel\_url* générée pour la route *paymentError*).



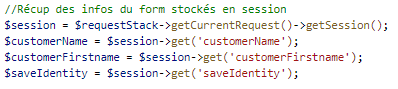
Enfin, l'utilisateur est redirigé vers l'URL de paiement fournie par Stripe.



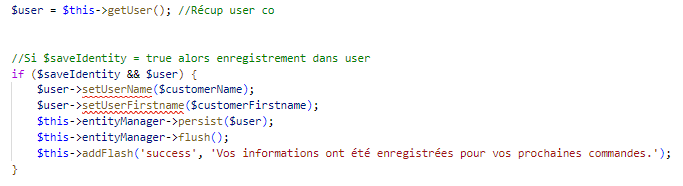
En cas d'échec du paiement (retour via *cancel\_url* et la route *paymentError*), l'utilisateur est redirigé vers la page du panier avec un message flash de type danger indiquant une erreur de paiement.



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



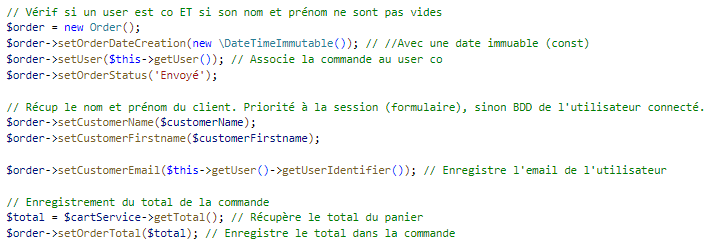
Si l'utilisateur a activé l'option d'enregistrement (*saveIdentity* est *true*), les propriétés *userName* et *userFirstname* de l'entité User sont mises à jour avec les données de session, puis *l'EntityManager* *Doctrine* est utilisé pour persister et enregistrer ces modifications en base de données.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



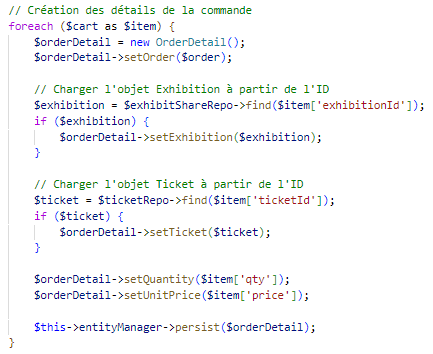
Ensuite, le contenu du panier est récupéré via le *CartService*.



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*.



*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*.



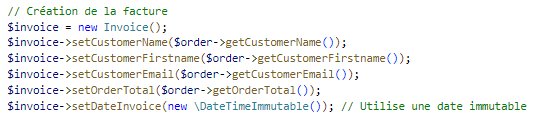
Ces entités *OrderDetail* sont ensuite persistées via *l'EntityManager*.



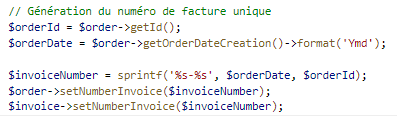
L'entité *Order* est rafraîchie (*$this->entityManager->refresh($order)*) afin de récupérer l'ID généré par la base de données, qui sera utilisé pour la création du numéro de facture unique.



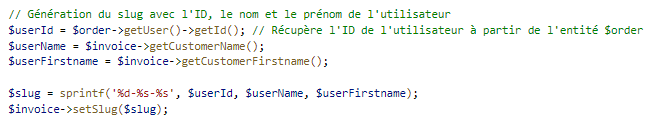
Une nouvelle instance de l'entité *Invoice* est instanciée et hydratée avec les informations de la commande : *customerName*, *customerFirstname*, *customerEmail*, *orderTotal* et *dateInvoice* (avec un objet \*DateTimeImmutable*).



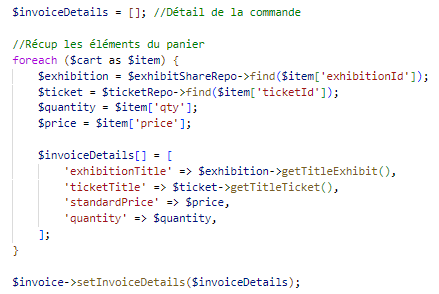
Un *numberInvoice* unique est généré en utilisant l'ID de la commande et la date de création de la commande (*orderDateCreation*). Ce numéro est ensuite attribué aux propriétés *numberInvoice* des entités *Order* et *Invoice*.



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*.



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.)



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



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.



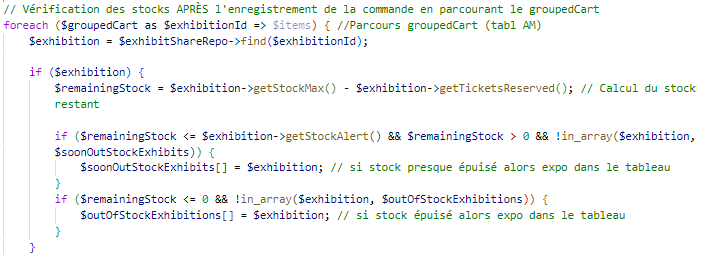
L'envoi d'une alerte de stock à l'administrateur est géré par le service *StockAlertEmailService* via sa méthode *sendStockAlertEmail*().

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



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*.



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.



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*).

