Diapo 17 :

Quand un utilisateur souhaite ajouter un ticket au panier, il clique sur l’icône +. Cela envoie une **requête HTTP de type GET** au contrôleur, qui exécute la méthode **AddTicketToCart()**

Diapo 18 :

Code à gauche :

Dans cette méthode j’injecte TicketRepository et ExhibitionShareRepository en paramétre ce qui respecte bien **l’encapsulation**.

Le contrôleur récupère les entités **Ticket** et **Exhibition** via leurs repositories respectifs en appelant la méthode **find(),** puis fait appel à un **service métier**, ici **CartService**, via la méthode **addCart()**.

Ce service est appelé dans le construct ce qui me permet de le réutiliser et d’isoler la logique métier liée au panier, respectant la séparation de responsabilité unique (solid).

Puis, selon l’origine, l’utilisateur est redirigé.

Code à droite :

Dans le service, la méthode **addCart(**) commence par récupérer les données de la session via une méthode getCart(). Elle enrichit ensuite les informations du ticket via une méthode personnalisée **findTicketDetails()** dans le repository.

Une clé unique est générée à partir de l’identifiant de l’exposition et du ticket, ce qui permet de gérer un **panier multi-expositions**. Si cette clé existe déjà, la quantité est incrémentée. Sinon, l’élément est ajouté au panier avec toutes ses données (ticket, prix, quantité, total).

Le panier et le total sont mis à jour et **sérialisés automatiquement dans la session.**

Diapo 19 :

Lorsque l’utilisateur a terminé ses achats, il accède à son panier pour procéder au paiement.

Si ce dernier n’a pas renseigné ses nom et prénom, un **FormType** lui est présenté et une **checkbox** lui permet d’enregistrer ou non son identité pour les futures commandes.

Il clique sur le bouton payer qui envoie une **requete http de type post** et la méthode stripeCheckout() sera exécutée.

Si le formulaire a été soumis et validé alors les informations sont enregistrées en session grâce aux setters.

Diapo 20 :

Ensuite, une **boucle itére** sur chaque ligne du panier afin de vérifier le stock : la quantité demandée est comparée au stock disponible **calculé dynamiquement**. Si un ticket est en rupture, une erreur est ajoutée à un **tableau d’erreur**. A la fin de l’analyse, une notification est envoyée à l’utilisateur lui indiquant le nom de l’exposition et le nombre de ticket restant.

Diapo 21 :

Si tout est valide, un **tableau est initié** pour préparer les données pour Stripe :

Pour cela, une boucle itère sur les informations requises par Stripe.

**L’API Stripe est configurée avec la clé secrète**, et les urls de sorties sont générées.

Ensuite une **session de paiement Stripe** est créée avec le tableau préalamblement rempli.

L’utilisateur est redirigé sur Stripe ou lui sera demandé ses coordonnées bancaires.

Diapo 22 :

Lorsque le paiement a réussi, la méthode StripeSuccess() est exécutée.

Si l’utilisateur a coché une case pour sauvegarder son identité, **l’entité User est mise à jour et persistée** avec Doctrine.

Diapo 23 :

A gauche

Ensuite, on **instancie l’objet de la classe Order** et on **l’hydrate** grâce aux **setters**.

A droite

On boucle sur chaque ligne du panier afin d’**instancier l’objet de la classe OrderDetails**.

Les **données désérialisées depuis la session**, on vérifie également que l’id en session est le meme qu’en base de données.

On sauvegarde et enregistre en base de données.

Diapo 24 :

Pour finir on instancie un nouvel objet de la classe invoice que l’on hydrate grâce aux setters. Le détail de la facture est **sérialisé sous forme de tableau associatif**.

Puis on sauvegarde et enregistre en base de données.