Très bien. Je vais analyser en profondeur la partie réservation de ton projet Cinephoria sur GitHub et établir un plan complet et détaillé pour une refonte totale, en respectant strictement le cahier des charges d'origine. J'exclurai les parties mobile et bureautique comme demandé, et je structurerai les étapes d'exécution pour que tu puisses les appliquer facilement.

Étape 1 : Analyse du code de réservation existant et éléments à supprimer

Avant de reconstruire la fonctionnalité, il faut identifier précisément **tous les composants liés à la réservation** dans le projet actuel afin de les supprimer proprement. Dans le code *RefactoCinephoriaWeb*, la réservation existante est probablement implémentée à travers plusieurs fichiers côté back-end et front-end :

- **Entités Doctrine** – Par exemple une entité `Reservation` (ou équivalent) représentant une réservation, possiblement une entité `Showtime` ou `Seance` (séance de cinéma) et peut-être une entité de liaison pour les places. Ces entités sont définies dans `src/Entity`.
- **Contrôleurs Symfony** – Cherchez un ou des contrôleurs dédiés à la réservation, par ex. `ReservationController.php` (ou nom similaire) dans `src/Controller`. Ce contrôleur gère probablement les routes de réservation existantes.
- **Templates Twig** – Les pages liées au parcours de réservation (choix du film, séance, etc.) sont définies dans `templates/`. Recherchez des fichiers Twig nommés par ex. `reservation/*.html.twig` ou tout template contenant le formulaire de réservation ou l'affichage des séances.
- **Scripts Front (Stimulus ou JS)** – Vérifiez dans `assets/controllers` (ou `assets/js`) la présence de contrôleurs Stimulus ou de fichiers JavaScript liés à la réservation (par exemple un contrôleur Stimulus pour la sélection de places ou le calcul du prix).
- **Routes et configuration** – Dans `config/routes.yaml` ou annotations de contrôleurs, identifiez les routes existantes de réservation (p. ex. `/reservation`, `/booking`, etc.) qu'il faudra retirer ou modifier.
- **Styles ou composants CSS** – Si la réservation avait des styles spécifiques (classes Tailwind, composants CSS), notez-les pour un nettoyage éventuel si non réutilisables.

Points de friction techniques identifiés : L'analyse du code existant peut révéler des problèmes qui justifient la reconstruction. Par exemple, une logique métier peut être dispersée dans le contrôleur ou la vue au lieu d'être centralisée, ce qui complique la maintenance. Des problèmes de **séparation des préoccupations** peuvent apparaître (logique de réservation directement dans les templates ou contrôleurs sans couche de service dédiée). On peut également rencontrer un couplage fort entre la réservation et d'autres parties du projet (ex. dépendances non souhaitées, difficultés à modifier le parcours utilisateur, etc.). Si l'implémentation actuelle ne respecte pas les bonnes pratiques Symfony (contrôleurs trop lourds, pas de tests, sécurité insuffisante), cela confirme la nécessité d'une refonte.

Enfin, assurez-vous qu'aucune autre partie du projet ne dépende de ces composants de réservation. Par exemple, si le menu de navigation ou la page profil utilisateur affiche des informations de réservation actuelles, notez ces interactions pour les adapter plus tard. L'objectif est de bien cerner l'empreinte du module de réservation existant avant de le supprimer.

Étape 2 : Suppression complète de l'ancienne fonctionnalité de réservation

Une fois les fichiers et dépendances repérés, on procède à une **suppression propre** :

- **Supprimer les fichiers back-end liés** : Retirez l'entité de réservation existante (`Reservation.php`) ainsi que les entités associées obsolètes (par exemple une entité `Ticket` ou `Seat` si présente et non réutilisable). Supprimez également le contrôleur de réservation (`ReservationController.php` ou similaire) ainsi que tout service ou repository dédié à l'ancienne réservation.
- **Nettoyer les templates Twig** : Supprimez les vues Twig de l'ancienne réservation. Par exemple, les fichiers dans `templates/reservation/` ou tout autre fichier affichant l'UI de réservation actuel. Vérifiez aussi le layout global (base Twig) ou la navbar : si un lien ou bouton « Réserver » existait, supprimez-le ou mettez-le en commentaire en attendant la nouvelle implémentation (pour éviter les liens cassés).
- **Retirer les routes** : Éliminez les entrées de routing YAML ou annotations correspondantes aux pages de réservation supprimées. Ceci évitera que l'application expose des URLs de réservation inexistantes. Par exemple, une route `/reservation` définie dans `routes.yaml` doit être commentée ou supprimée, de même que toute route vers un ancien contrôleur de réservation.
- **Scripts front** : Supprimez ou désactivez les fichiers JS/Stimulation liés. Par exemple, si un contrôleur Stimulus nommé `reservation_controller.js` existe, retirez son import dans `importmap.php` ou dans le bootstrap Stimulus, et supprimez le fichier. Idem pour tout plugin ou script de sélection de place antérieur (par ex. un fichier JS custom).
- **Base de données** : Préparez une **migration Doctrine** pour supprimer les tables obsolètes de réservation. Par exemple, si une table `reservation` existait avec une relation vers `user` et `showtime`, et qu'on décide de repartir sur de nouvelles bases, créez une migration qui drop cette table et éventuellement les tables liées (par ex. `ticket`ou `seat_reserved`). ⚠️ **Important** : assurez-vous que la suppression de table ne provoque pas d'erreurs d'intégrité référentielle (vérifiez les contraintes de clé étrangère). Au besoin, ajustez l'ordre des drops ou utilisez `ON DELETE CASCADE` pour nettoyer les dépendances.
- **Vérifications** : Après suppression, lancez l'application (ou les tests existants) pour s'assurer qu'aucune route ne référence encore le module de réservation supprimé. Naviguez sur le site pour vérifier que le menu ou les pages utilisateur ne comportent plus de fonctionnalités brisées. Le projet doit fonctionner sans la moindre trace de l'ancienne réservation avant de commencer la reconstruction.

En procédant ainsi, on **assainit le codebase** et on évite les conflits entre l'ancienne et la nouvelle implémentation. Le code est allégé des composants inutiles, ce qui facilite la lecture et la mise en place du nouveau module. N'hésitez pas à commit ces suppressions séparément ("remove old reservation module") pour conserver un historique clair, au cas où il faudrait consulter l'ancienne logique durant le développement de la nouvelle.

Étape 3 : Conception du schéma de base de données pour la nouvelle réservation

La reconstruction débute par la **conception de la base de données** et des entités métier nécessaires. D'après le cahier des charges, la réservation implique plusieurs entités reliées : Cinéma, Film, Séance, Place (ou du moins nombre de places) et Réservation. On conçoit le schéma SQL en respectant ces besoins :

- **Entité Cinéma** : si ce n'est pas déjà le cas dans le projet, créer une table/entité `Cinema` contenant les cinémas du réseau Cinéphoria. Cette entité comporte des champs tels que `id`, `nom` (ou `ville`/`adresse` si nécessaire). Chaque cinéma possédera plusieurs salles et propose plusieurs films.
- **Entité Salle (Auditorium)** : chaque cinéma a une ou plusieurs salles. Créez une entité `Salle` avec `id`, `nom` ou numéro de salle, capacité (`capacite` nombre de places). Elle possède une relation Many-to-One vers `Cinema`(plusieurs salles par cinéma). Cela permettra de connaître le nombre total de sièges disponibles par salle pour les séances. *(Si le cahier des charges ne distingue pas plusieurs salles par cinéma, on peut simplifier en incluant un champ `salle` dans l'entité Séance. Cependant, avoir une entité dédiée rend l'application évolutive.)*
- **Entité Film** : il existe probablement déjà une entité ou du moins des données de films (via API ou base interne). Si une entité `Film` existe (contenant titre, description, etc.), on la réutilise. Sinon, créez-en une minimale avec `id`, `titre` (et éventuellement durée, genre…). Les séances feront référence aux films.
- **Entité Séance** : c'est une pièce centrale du système de réservation. Créez l'entité `Seance` (ou `Showtime`) avec les champs : `id`, date et heure de la séance (`dateHeure` DateTime), qualité (`qualite` – par ex. *2D, 3D, VO, VF*… stocké en texte ou enum), relation Many-to-One vers `Film`, relation Many-to-One vers `Salle` (ou vers `Cinema` si on n'a pas de salles multiples). Ajoutez un champ `placesDisponibles` (nombre de places restantes). **Important**: ce champ peut être calculé à partir de la capacité de la salle et des réservations faites, mais l'inclure facilite l'affichage rapide du nombre de places disponibles. Alternativement, stockez uniquement la capacité via la salle et calculez les places restantes dynamiquement (somme des réservations déjà faites) pour éviter la redondance. Vous pouvez choisir l'approche en fonction de la simplicité vs. performance. (Dans un système plus complexe, on pourrait lier chaque siège individuellement, mais le cahier des charges semble se focaliser sur le nombre de places disponibles).
- **Entité Réservation** : créez l'entité `Reservation` pour représenter la réservation d'un ou plusieurs sièges par un utilisateur. Champs principaux : `id`, nombre de places réservées (`nombrePlaces`), prix total, horodatage de réservation (`dateReservation`). Relations : Many-to-One vers `Seance` (chaque réservation concerne une séance précise), Many-to-One vers l'entité `User` (car la réservation est faite par un utilisateur inscrit). Cette table stocke chaque commande de l'utilisateur.
- *(Optionnel)* **Entité Ticket/Billet** : le besoin ne mentionne pas de billet individuel ni de sélection de sièges précis, seulement le nombre de places. On peut donc ne **pas** créer de table séparée pour chaque place réservée, simplifiant ainsi le modèle. Chaque réservation aura un champ nombrePlaces > 1 le cas échéant. Si plus tard on devait attribuer des numéros de siège spécifiques, on pourrait introduire une table de jointure `ReservationPlace`associant Réservation et des identifiants de sièges, mais ce niveau de détail semble hors du périmètre actuel.

Une fois ces entités définies, déterminez les **contraintes et index** SQL :

- Clés étrangères appropriées (par ex. `Reservation.user_id` vers `User.id`, `Reservation.seance_id` vers `Seance.id`, etc., avec on delete cascade si on supprime une séance ou un user, selon les règles métier).
- Contrainte d'unicité éventuelle : on peut imposer qu'une même place ne soit vendue qu'une fois par séance. Dans notre modèle simplifié par nombre de places, cela revient à vérifier que le total des réservations d'une séance n'excède pas sa capacité. Une contrainte d'unicité au niveau place n'est pas nécessaire ici (on n'a pas de table Place), mais on pourrait imaginer une contrainte au niveau de la réservation qui rejette une réservation si `nombrePlaces > placesDisponibles`. Cela se gèrera plutôt côté logique applicative.
- Index : index sur `Seance.cinema_id` ou `Seance.film_id` pour accélérer la recherche des séances par cinéma ou film, index sur `Reservation.user_id` pour lister rapidement les réservations d'un utilisateur, etc.

Côté **migration** : utilisez la console Symfony pour générer une migration après avoir codé les entités (commande `php bin/console make:migration`). Vérifiez le SQL généré – il doit créer les nouvelles tables (`cinema, salle, film, seance, reservation` etc.) et leurs contraintes. Exécutez ensuite `php bin/console doctrine:migrations:migrate`pour appliquer le schéma.

NB: Si certaines entités existaient déjà (par ex. `Cinema` ou `Film`), adaptez plutôt que recréer. Par exemple, si l'application gérait déjà une liste de films via une API MongoDB, il faudra décider de comment lier cela avec les séances SQL. On peut stocker seulement un *filmId* TMDB dans `Seance` et aller chercher les détails via l'API ou base Mongo, ou bien importer les films nécessaires en base SQL. Ce point dépend de l'architecture existante (dans l'ECF mentionné, possiblement les films sont gérés via une API TMDB et MongoDB). Pour notre plan, on suppose qu'une entité Film SQL existe ou sera créée pour simplifier les relations.

En résumé, cette conception de base de données respecte les besoins du domaine **cinéma** : plusieurs cinémas, contenant des salles, projetant plusieurs films via des séances, avec un certain nombre de places par séance, et des utilisateurs pouvant réserver ces places. Ce modèle suit les principes courants d'un système de réservation de cinéma (une salle avec X sièges, chaque séance vend chaque siège au plus une fois). Une fois validé, il servira de fondation pour la couche métier et la couche présentation.

Étape 4 : Architecture back-end en couches et logique métier Symfony

Avec le schéma de données en place, on structure l'implémentation côté serveur en respectant l'**architecture en couches** de Symfony 7 et les bonnes pratiques de conception. L'objectif est de séparer clairement les responsabilités entre les différents composants : **Contrôleurs**, **Services** et **Dépôts (Repositories)**, conformément aux principes SOLID. On vise des **contrôleurs les plus légers possible**, avec la logique métier déléguée à des services réutilisables. Voici le plan :

- **Repositories & Entités** : Utilisez les *Repository* Doctrine pour encapsuler l'accès aux données. Symfony génère par défaut une classe repository pour chaque entité

(ex. `SeanceRepository`, `ReservationRepository`). On ajoutera dans ces classes des méthodes de recherche spécifiques nécessaires au parcours utilisateur. Par ex.: `findByCinemaAndFilm(cinema, film)` dans `SeanceRepository` pour lister les séances d'un film dans un cinéma donné,
ou `countReservationsForSeance(seance)` pour obtenir le nombre total de places déjà réservées sur une séance (utile pour calculer les places restantes si on ne stocke pas ce champ).

- **Services métier (Business services)** : Créez un service dédié à la réservation, par ex. `ReservationService` (ou `BookingManager`). Ce service contiendra la logique métier centrale du processus de réservation, utilisable par les contrôleurs et potentiellement ailleurs. Par exemple, des méthodes possibles :
  - `listMoviesForCinema(cinema)` – retourne la liste des films ayant des séances à venir dans un cinéma (en s'appuyant sur le repository Seance).
  - `listSeances(cinema, film)` – retourne les séances disponibles (non passées) pour un film donné dans un cinéma, avec éventuellement un filtrage par date.
  - `calculatePrice(seance, nombrePlaces)` – calcule le prix total pour une séance en fonction de la qualité et du nombre de places. La fonction peut appliquer la règle métier (par ex. tarif de base * nombre, avec supplément si `seance.qualite == "3D"`, etc.).
  - `bookSeats(user, seance, nombrePlaces)` – réalise la réservation : vérifie les conditions (utilisateur connecté, nombrePlaces ≤ places disponibles), décrémente le stock de places de la séance, crée et persiste l'entité Reservation correspondante, retourne le résultat (réservation réussie ou exception/erreur si échec).
    Ce service utilise les repositories pour lire/écrire en base. En isolant ces règles ici, on pourra facilement les réutiliser et les tester indépendamment des contrôleurs.
- **Contrôleurs Symfony** : Mettez en place des contrôleurs RESTful ou orientés pages pour chaque étape du parcours utilisateur (voir étape 5). Vous pouvez opter pour un **contrôleur unique** `ReservationController`gérant toutes les étapes de réservation, ou bien segmenter en plusieurs contrôleurs logiques (par ex. `CinemaController` pour choisir le cinéma et le film, `SeanceController` pour afficher les séances, `ReservationController` pour la validation finale). Dans les deux cas, faites en sorte que chaque méthode de contrôleur soit concise : elle fera appel aux services pour la logique. Par exemple, la méthode pour afficher les séances d'un film dans un cinéma fera simplement : appeler `listSeances(cinema, film)` du service et passer le résultat à la vue. Un **contrôleur "léger"** évite la duplication de code et se concentre sur l'agrégation des données et le déclenchement des bonnes méthodes.
- **Validation et Exceptions** : Le service de réservation pourra lever des exceptions personnalisées (par ex. `PlacesInsuffisantesException`) si les règles ne sont pas respectées, que le contrôleur pourra attraper pour renvoyer un message d'erreur approprié à l'utilisateur (voir parcours utilisateur). De même, utilisez les **Validation Constraints** de Symfony sur les entités/formulaires (par ex. `@Assert\Positive` sur `Reservation.nombrePlaces`, contrainte personnalisée pour vérifier qu'on ne dépasse pas le max de la séance). La logique de **validation des données** doit être appliquée à la fois côté front (JavaScript) pour l'expérience utilisateur, et côté back (service ou entité) pour la robustesse (on ne fait pas confiance au seul front).

- **Couche de présentation (Twig)** : Bien que ce soit du front, mentionnons que le contrôleur passera aux vues uniquement les données nécessaires (par ex. liste des films, détails de la séance, etc.) – aucune logique de calcul n'y sera faite. Le Twig doit rester simple (affichage de données et formulaires).

En suivant cette architecture, on garantit une bonne maintenabilité. Par exemple, si l'on doit réutiliser le calcul de prix ailleurs, on a une seule fonction dans le service (DRY principle). Si plus tard on ajoute une API mobile, le service pourra être réutilisé pour effectuer les réservations sans dupliquer la logique. Cette séparation claire (Controller -> Service -> Repository) est conforme aux conseils de la communauté Symfony, qui encourage le respect strict du découpage en couches avec SOLID. Ainsi, *le contrôleur se consacre à l'interface utilisateur, le service à la logique métier, et le repository à la persistance*, conformément aux bonnes pratiques.

Étape 5 : Conception de l'interface utilisateur Web (Tailwind CSS) et interactions (Stimulus)

En parallèle de la couche back-end, on doit préparer l'**interface utilisateur** pour le parcours de réservation. L'UI sera développée en Twig avec Tailwind CSS pour le style, et enrichie de comportements dynamiques avec Stimulus (framework JavaScript intégré via Symfony UX). L'objectif est de réaliser une expérience utilisateur fluide pour l'US4, en respectant les maquettes Figma fournies (si disponibles). Chaque étape du parcours correspond à une vue claire ; on les détaille ci-dessous, avec les aspects UX et techniques :

- **Choix du cinéma** : La première page du parcours permettra à l'usager de sélectionner le cinéma souhaité. D'après l'US4, c'est le point de départ. Implémentez une vue Twig (par ex. `reservation/choose_cinema.html.twig`) qui liste tous les cinémas disponibles. Utilisez le *controller* pour passer la liste des cinémas (via `CinemaRepository::findAll()` ou un service). En termes de design Tailwind, on peut afficher chaque cinéma sous forme de carte ou d'un bouton liste. Par exemple, une grille de cartes avec le nom du cinéma, son lieu, et un bouton « Choisir ». **Stimulus** : Pour cette étape, l'interactivité peut être minimale (un simple clic redirige vers l'étape suivante). Cependant, si on voulait améliorer l'UX, on pourrait intégrer Stimulus pour, par exemple, filtrer les cinémas par ville en direct (si de nombreuses entrées). Dans le contexte initial, probablement pas nécessaire – une simple liste cliquable suffit.
- **Choix du film** : Après sélection du cinéma, l'utilisateur arrive à la page de choix du film dans ce cinéma. Route envisagée : `/reservation/cinema/{id}/films`. Le contrôleur va utiliser le service (ex. `listMoviesForCinema`) pour obtenir les films actuellement programmés dans ce cinéma (on ne montre que les films pour lesquels il y a des séances ouvertes à la réservation). La vue `reservation/choose_film.html.twig` affichera ces films, par exemple sous forme d'affiches ou titres avec un bouton « Voir les séances ». Stylisez avec Tailwind (cards ou simples lignes). **Interactions Stimulus** : on peut rendre la navigation plus dynamique. Par exemple, plutôt que d'avoir une page séparée, on pourrait en une seule page afficher d'abord la liste des films dès qu'un cinéma est choisi (via une requête AJAX). Un contrôleur Stimulus pourrait écouter le clic sur un cinéma et charger la liste des films via fetch API (pointant vers une API interne genre `/api/cinema/{id}/films`). Cela éviterait un rechargement complet.

Néanmoins, pour simplicité, on peut aussi faire page par page. Si vous avez le temps, implémentez ce raffinement : un **Stimulus controller** (par ex. `CinemaController` front-end) qui sur le choix d'un cinéma va remplir une liste de films sans reload. Cela utilise Symfony UX ou un endpoint JSON renvoyant les films. *(Cette amélioration n'ajoute pas de fonctionnalité métier, juste du confort UX, donc reste optionnelle.)*

- **Affichage des séances disponibles** : Une fois le film sélectionné, l'utilisateur doit voir toutes les séances correspondantes dans le cinéma choisi.
  Route : `/reservation/cinema/{id}/film/{idFilm}/seances` (ou une route plus simple du type `/reservation/seances?cinema=X&film=Y`). Le contrôleur récupère via le service `listSeances(cinema, film)` la liste des séances à venir. La vue Twig `reservation/list_seances.html.twig`va lister chaque séance avec **qualité**, **horaire** et **salle**. Par exemple : « Mardi 5 juin 20:30 – Salle 1 – **3D** – 50 places dispo ». On peut utiliser un composant Tailwind pour les listes ou des tableaux. Il est judicieux d'**indiquer le nombre de places disponibles** pour chaque séance, comme suggéré par le cahier des charges ("Vérification du nombre de places disponibles"). Ainsi, l'utilisateur peut évaluer d'un coup d'œil la disponibilité (et éviter de cliquer sur une séance complète). Si une séance est complète (0 places), affichez-le clairement (par ex. grisé, label "Complet" et désactivez le bouton de réservation).
    - **Stimulus éventuel** : la liste de séances en soi n'exige pas de JS, c'est purement informatif + lien. Cependant, vous pouvez améliorer l'ergonomie en permettant, par exemple, un **filtrage** par version/qualité (montrer uniquement 3D ou VO via un toggle Stimulus), ou un tri par horaire. Ceci n'est pas explicitement demandé, donc à votre discrétion.
- **Sélection du nombre de places** : En cliquant sur "Réserver" sur une séance donnée, on passe à l'étape de choix des places.
  Route : `/reservation/seance/{id}/choix` (contrôlée par ex. `ReservationController::choixPlaces($seanceId)`). **Sécurité** : cette route doit être **protégée par authentification**, car l'US4 impose un compte pour réserver. On pourra configurer cette protection via l'annotation `@IsGranted("ROLE_USER")` sur le contrôleur ou via `security.yaml` (access_control pour le pattern `/reservation/**` pour les utilisateurs connectés). Ainsi, si un utilisateur non connecté tente d'y accéder, il sera redirigé vers la page de login automatiquement. *(Symfony Security gère cela simplement : on définit que la page nécessite d'être authentifié, toute requête anonyme sera renvoyée vers le formulaire de login)*.
    - **Formulaire** : Le contrôleur charge la séance correspondante (404 si ID invalide ou séance introuvable). Il vérifie aussi que la séance n'est pas passée (on ne réserve pas dans le passé) et qu'il reste au moins 1 place. Ensuite, il passe à la vue `reservation/choose_places.html.twig` les informations de la séance (film, cinéma, horaire, qualité, places dispo) et un formulaire pour saisir le nombre de places à réserver. Ce formulaire comporte un champ numérique `nombrePlaces`. Vous pouvez limiter ce champ de 1 jusqu'au maximum disponible (`seance.placesDisponibles`). Utilisez les **composants Tailwind** pour le style du formulaire (par ex. champs avec bordures, etc.).
    - **Calcul du prix** : Juste en dessous du champ, affichez le prix total calculé en fonction du nombre de places sélectionné et de la qualité. Par exemple : « Total : 30 € » pour 3 places en 2D à 10€/place, ou « Total : 36 € » si 3D à 12€/place. La **logique de tarification** peut être codée côté front et côté back :

        - Côté back, le service `calculatePrice(seance, nb)` peut être appelé au moment où l'utilisateur soumettra (pour stocker le montant dans la réservation).
        - Côté front, pour améliorer l'UX, on utilise **Stimulus** afin de mettre à jour le prix total dynamiquement à chaque changement du nombre de places. Créez un contrôleur Stimulus (par ex. `price_controller.js`) associé au champ nombre de places. Ce contrôleur pourra soit recalculer en JS pur (ex: il connaît le prix unitaire de la séance – qu'on pourrait passer en data attribute, ou récupère via dataset les infos qualité pour appliquer un tarif), soit interroger une petite API (ex: route `/api/calcPrice?seance=X&qty=Y`). Le plus simple est de faire le calcul en JS localement si la règle tarifaire est simple (ex: tarif2D = 10€, tarif3D = 12€). Ainsi, dès que l'utilisateur modifie le nombre, le contrôleur Stimulus fait `total = tarifUnitaire * nombre` et affiche le résultat formatté. Cela offre une réactivité instantanée sans recharger la page. *(Cette approche suit la recommandation Symfony d'utiliser Stimulus pour du JS structuré dans une appli server-side.)*
    - **Validation front** : Ajoutez avec Stimulus une vérification immédiate : si l'utilisateur entre un nombre supérieur au disponible, afficher un message d'erreur (et éventuellement désactiver le bouton). Bien sûr, la validation finale se fera au back-end aussi, mais cette aide côté client améliore l'expérience.
    - **CTA Réserver** : Le formulaire une fois rempli propose un bouton « Confirmer la réservation ». Ce sera une soumission POST vers, par exemple, la route `/reservation/seance/{id}/valider` gérée par `ReservationController::valider($seanceId)` (méthode séparée). Incluez un **token CSRF** dans le form pour sécuriser la requête (Symfony Forms le fait automatiquement si vous utilisez le composant Form; sinon utilisez la fonction Twig `{{ csrf_token('reservation') }}` et vérifiez-le dans le contrôleur). Symfony protège automatiquement les formulaires contre la CSRF si configuré, il suffit de rendre le champ token dans la vue et d'appeler `isCsrfTokenValid()` côté contrôleur.
- **Confirmation de la réservation** : Lorsque l'utilisateur soumet le formulaire, le contrôleur `valider` reçoit les données. Il vérifiera de nouveau les règles métier via le service : utilisateur connecté (normalement garanti par la sécurité de route), **données valides** (ex. le nombre de places soumis via le form ne dépasse pas `seance.placesDisponibles` – malgré nos contrôles front, il faut revalider côté serveur). Utilisez les contraintes de validation Symfony ou faites manuellement : si l'utilisateur a triché en modifiant le HTML pour demander plus de places, le service le détectera et lèvera une exception ou retournera une erreur. En cas d'erreur (plus de places disponibles, nombre invalide, etc.), renvoyez la vue de sélection avec un **message d'erreur** approprié (par ex. "Nombre de places invalide ou insuffisant"). Sinon, si tout est bon, le service `bookSeats` :
    - Crée la réservation en base (`Reservation` reliée au user et séance) en enregistrant le nombre de places et le prix total calculé.
    - Met à jour le nombre de places disponibles de la séance (décrémenter ou recalculer `placesDisponibles` = ancien `placesDisponibles` - nombre réservé, ou si on ne stocke pas ce champ, ce calcul se fait à l'affichage seulement). Ici on assurera qu'on ne tombe pas en négatif. Idéalement, enveloppez ces opérations dans une transaction pour garantir la cohérence (begin transaction, persister réservation, mettre à jour séance, commit).

Doctrine gère la transaction sur flush globale généralement, mais dans des cas de concurrence réelle, il faudrait verrouiller la ligne séance en base (pessimistic lock) pour éviter deux réservations simultanées d'excéder la capacité. Étant un projet pédagogique, mentionnons simplement que le **contrôle métier** empêchera le dépassement et que l'on considère ce cas.
    - Si l'enregistrement réussit, redirige l'utilisateur vers la page de confirmation.
- **Affichage post-confirmation dans l'espace utilisateur** : Plutôt que de créer une page isolée "Réservation confirmée" (qui serait possible, avec un récapitulatif), il est demandé de la faire apparaître dans l'**espace personnel** de l'utilisateur. Donc, on peut choisir de rediriger vers la page profil/compte de l'utilisateur (par ex. route `/mon-compte` déjà existante) avec un ancrage ou un message flash. Cependant, il est plus précis de supposer que l'espace utilisateur contient une section "Mes réservations". Dans ce cas, après la réservation, utilisez `addFlash('success', 'Votre réservation pour X places à la séance Y a bien été confirmée.')` puis faites `return $this->redirectToRoute('user_reservations')`. Vous devrez implémenter dans le **Tableau de bord utilisateur** (peut-être géré par un `UserController` ou `ProfileController` existant) l'affichage de la liste des réservations de l'utilisateur connecté. Si cette section n'existe pas, créez-la : une méthode `mesReservations()` qui utilise `ReservationRepository->findBy(['user' => $this->getUser()])` pour obtenir les réservations de l'utilisateur, classées par date. La vue Twig correspondante (ex. `user/reservations.html.twig`) listera les réservations avec leurs détails (film, cinéma, horaire, nombre de places, qualité, prix). Ainsi, l'utilisateur peut voir immédiatement sa nouvelle réservation apparaître dans cette liste, confirmant son succès.
    - **Design** : utilisez Tailwind pour mettre en forme ce tableau ou cette liste de réservations (par ex. un tableau avec colonnes "Film", "Cinéma", "Séance", "Places", "Total payé"). Si un design Figma existe pour le dashboard utilisateur, alignez-vous dessus.
    - **Interactions** : en général, la consultation de la liste n'exige pas de Stimulus. On peut éventuellement permettre un tri ou une annulation de réservation via du JS, mais ce n'est pas mentionné.

En suivant ce parcours étape par étape, on recouvre l'intégralité de l'US4. Chaque écran est pensé pour être clair et guidé. Le **design TailwindCSS** assure une UI moderne et homogène avec le reste du site (puisque Tailwind est utilisé globalement, les nouvelles pages s'intégreront visuellement sans effort majeur – utilisez les classes utilitaires existantes et respectez éventuellement le thème configuré dans `tailwind.config.js`). Les prototypes Figma, s'ils sont fournis, devraient donner une idée précise de la mise en page (par exemple, disposition des cartes de films, style des boutons…). Reproduisez-les fidèlement en utilisant la grille et les espaces Tailwind.

Du côté **JavaScript/Stimulus**, on a identifié des points clefs pour améliorer l'expérience : mise à jour dynamique du prix, contrôle de saisie du nombre de places, et potentiellement chargement asynchrone des films ou filtre de séances. Stimulus étant recommandé dans Symfony UX, l'intégrer ainsi permet de garder du JS organisé par fonctionnalité, facilement maintenable. Chaque contrôleur Stimulus doit être discret, ne faisant qu'enrichir l'UX sans dépendre du serveur outre mesure. Veillez à bien **documenter dans le HTML** via les attributs `data-controller`, `data-action` et `data-*` pour que les intentions soient claires

(ex: `<div data-controller="price" data-price-base="10" data-price-supplement="2">`).

En somme, cette étape aboutit à un **ensemble de pages web** bien structurées, reliées par des routes Symfony, avec une navigation aisée et une interface claire pour l'utilisateur. Le tout en conservant la cohérence visuelle (via Tailwind) et la fluidité (grâce à Stimulus pour l'interactif).

Étape 6 : Tests unitaires et tests end-to-end (Cypress)

Avant le déploiement, il est crucial d'écrire des tests pour assurer la fiabilité de la nouvelle fonctionnalité de réservation. On couvrira à la fois des **tests unitaires** (PHPUnit) pour la logique métier et des **tests end-to-end** (E2E) avec Cypress pour valider le parcours utilisateur complet dans un navigateur réel. Ceci garantira que la fonctionnalité correspond au cahier des charges et évitera les régressions.

Tests unitaires (PHPUnit) : Concentrez-vous sur les composants critiques du back-end, en isolant la logique :

- *Service de Réservation* : testez les méthodes du `ReservationService`. Par exemple, testez `calculatePrice()` avec différents cas : séance en 2D vs 3D, nombre de places = 1, plusieurs places, vérifier que le calcul est correct (ex. 3 places 3D = 3 * tarif3D). Testez `bookSeats()` : préparez une séance fictive avec X places dispos, appelez bookSeats avec un nombre valide et assurez-vous qu'il retourne une réservation avec les bonnes infos, que le stock de places est mis à jour correctement. Testez aussi les cas d'erreur : demander plus de places que dispo doit soit lever une exception soit retourner une indication d'échec – le test doit vérifier ce comportement (par ex. s'attendre à une Exception spécifique).
- *Validation* : si vous avez créé des contraintes personnalisées (par ex. Validator de disponibilité), testez-les séparément. Idem pour toute fonction utilitaire.
- *Repository* : Les repository en tant que tels utilisent Doctrine (déjà testée), mais si vous avez ajouté des méthodes custom (ex: `findMoviesByCinema`), vous pouvez écrire un test en mode *integration* avec une petite base de données de test (ou en moquant Doctrine) pour vérifier que la requête retourne ce qui est attendu. Cependant, souvent on fait plutôt confiance à Doctrine et on couvre cela via les tests fonctionnels/E2E.
- *Contrôleurs* : Pour les contrôleurs, on peut écrire des **tests fonctionnels Symfony** (via WebTestCase) pour simuler des requêtes HTTP sans navigateur. Par exemple, tester que l'accès à la page de sélection des places sans être authentifié renvoie une redirection vers /login (Symfony gère ça, mais on peut l'asserter). Ou tester que le fait de soumettre une réservation avec un CSRF token invalide est rejeté (403). Ces tests garantissent que la sécurité est bien en place.

Les tests unitaires doivent couvrir les scénarios normaux et d'exception, pour prouver que la logique métier est robuste. Chaque bug trouvé doit entraîner l'ajout d'un test évitant sa réapparition. L'objectif est d'atteindre une bonne couverture sur le module de réservation.

Tests end-to-end (Cypress) : Cypress permettra de **simuler un vrai utilisateur** naviguant sur l'application via un navigateur, et de vérifier que tout le parcours (front + back)

fonctionne et qu'aucun élément visuel ou enchaînement n'est cassé. Cypress est un outil très adapté car il simule les interactions utilisateur de bout en bout et vérifie l'interface dans son ensemble. On écrira des scénarios couvrant l'US4 :

- *Scénario de réservation réussie* :
    1. **Login utilisateur de test** – Utilisez un compte de test (créé via les fixtures ou la base d'essai, ex. `visitor@example.org` comme dans le jeu d'essai ECF). Dans Cypress, ouvrez la page de login, entrez les identifiants, soumettez et vérifiez que la connexion réussit (par exemple, en testant que le nom de l'utilisateur apparaît dans le header ou qu'on est redirigé sur la page d'accueil).
    2. **Choix du cinéma** – Naviguez sur la page de choix du cinéma (URL de départ du parcours). Vérifiez que la liste des cinémas s'affiche. Simulez un clic sur un des cinémas.
    3. **Choix du film** – Après le clic, Cypress doit constater l'affichage de la liste des films pour le cinéma sélectionné. Vérifiez que la page contient bien des éléments correspondants aux films attendus (par ex. le titre d'un film connu présent en base). Cliquez sur un film.
    4. **Liste des séances** – La page des séances doit apparaître. Vérifiez la présence de plusieurs horaires, et notamment que pour chaque séance on voit la salle, la qualité et le nombre de places dispos. Si certaines séances sont « Complet » (0 places), assurez-vous que Cypress les voit marquées comme telles et éventuellement non cliquables. Choisissez une séance qui a des places disponibles en cliquant sur son bouton "Réserver" ou lien.
    5. **Choix des places** – Sur la page de sélection, Cypress va interagir avec le formulaire : vérifier que le champ du nombre de places existe et est borné (par exemple, essayer d'entrer un nombre trop grand et voir si l'UI réagit). Simuler la sélection de, disons, 2 places. Vérifier que le prix total affiché se met bien à jour (Cypress peut obtenir le texte du DOM du champ du total et le comparer à la valeur attendue, par ex. "Total: 24 €" si 2 places en 3D à 12€).
    6. **Soumission** – Cliquer sur le bouton de confirmation. Cypress doit alors vérifier que : (a) une requête réseau est bien envoyée (il peut intercepter l'appel POST et vérifier le code réponse 302 de redirection), (b) que l'application redirige vers l'espace utilisateur (URL du profil), et (c) que sur la page du profil, la nouvelle réservation apparaît bien dans la liste "Mes réservations". On peut faire chercher par Cypress une ligne de tableau contenant par exemple le nom du film et l'horaire qu'on vient de réserver. Si c'est présent, le test confirme que la réservation a été enregistrée et affichée.
    7. **État en base** – Optionnellement, on peut coupler Cypress avec une vérification back-end (ou simplement faire un appel API interne) pour s'assurer que la table Reservation a bien une nouvelle entrée associée à l'utilisateur de test. Mais si l'affichage UI montre la résa, c'est suffisant.
- *Scénario d'accès non authentifié* : Lancez un test où l'utilisateur *non connecté* essaie d'accéder directement à l'URL de sélection des places d'une séance. Cypress devrait être redirigé vers la page de login. On valide ainsi la contrainte "obligation de compte utilisateur" du cahier des charges.
- *Scénario d'erreur de disponibilité* : On peut forcer un cas où l'utilisateur demande trop de places. Par exemple, via un petit script SQL avant le test, définissez une séance avec seulement 1 place disponible. Ensuite, dans Cypress, faites le parcours sur cette séance en essayant de réserver 2 places. On attend que l'UI empêche la soumission ou que le serveur réponde avec un message d'erreur. Vérifiez que l'application affiche

bien “places insuffisantes” (ou le message que vous avez prévu) et reste sur la même page sans valider. Ce test garantit la **vérification du nombre de places disponibles** côté front et back.

Cypress offre une visualisation étape par étape de ces interactions, utile pour débugger. Ces tests E2E vérifient **le bon enchaînement des pages et la cohérence des données affichées**, ce qui complète parfaitement les tests unitaires. En automatisant ainsi les parcours utilisateur, on s’assure que les exigences fonctionnelles (US4) sont pleinement réalisées.

En résumé, l’ensemble de la suite de tests doit couvrir : calculs de prix, règles de réservation (unitaire), parcours complet normal, parcours erreur, et contraintes de sécurité. Ceci correspond aux critères d’acceptation du cahier des charges. N’oubliez pas d’intégrer ces tests à votre pipeline d’intégration continue si disponible, pour exécution automatique. Un module aussi critique que la réservation doit rester fiable à chaque modification du code.

Étape 7 : Sécurité, authentification et validation des données

La sécurité est un aspect transversal qu’il convient de soigner dans cette refonte. Plusieurs axes : **contrôles d’accès**, **protection CSRF**, **validation des entrées utilisateur**, et bonnes pratiques générales pour éviter les failles.

- **Authentification obligatoire** : Comme spécifié, **un compte utilisateur est requis pour valider** une réservation. Cela se traduit, techniquement, par la nécessité que l’utilisateur soit authentifié (role USER) avant d’accéder à la phase finale. Pour le garantir, configurez vos routes ou méthodes de contrôleur en conséquence. Deux approches :
    1. *Via le firewall/security.yaml* : définir dans `security.yaml` un access_control qui restreint l’accès aux URL de réservation. Par ex:

```
access_control:
    - { path: '^/reservation', roles: IS_AUTHENTICATED_FULLY }
```

       Ainsi toute URL commençant par /reservation nécessitera un utilisateur connecté (FULLY, c.-à-d. non juste “remember me”). Symfony redirigera automatiquement vers la route de login configurée si non satisfait.

    4. *Via annotations* : Sur le contrôleur ou les actions critiques, ajouter `@IsGranted("ROLE_USER")` (ou `@Security("is_granted('IS_AUTHENTICATED_FULLY')")`). Cela jette un AccessDeniedException si non logué, que le firewall interceptera pareillement pour renvoyer au login.
       De plus, dans les templates Twig, on peut améliorer l’UX en cachant ou désactivant les boutons de réservation tant que l’utilisateur n’est pas logué, avec des conditions `{% if app.user %}...{% else %} (invitez à se connecter) {% endif %}`. Mais même sans cela, le back-end doit de toute façon bloquer.
- **CSRF protection** : Toutes les actions de formulaire (comme la soumission de réservation) doivent être protégées contre les attaques CSRF. Symfony Forms active par défaut cette protection, donc si vous utilisez une `FormType`pour le nombre de places, vous n’avez rien de plus à faire (un champ `_token` sera inclus et vérifié

automatiquement). Si vous n'utilisez pas de FormType Symfony (par ex., vous avez construit le `<form>`manuellement en Twig), alors ajoutez manuellement un champ hidden avec un token CSRF et vérifiez-le dans le contrôleur. Exemple Twig :

- `<input type="hidden" name="_token" value="{{ csrf_token('reservation' ~ seance.id) }}">`

et côté PHP :

```
if (!$this->isCsrfTokenValid('reservation'.$seance->getId(), $request->request->get('_token'))) {
    throw new InvalidCsrfTokenException('CSRF token invalid');
}
```

Ceci empêche qu'un tiers puisse forger un formulaire de réservation soumis à l'insu de l'utilisateur.

- **Validation des données (Back-end)** : Bien que l'UI limite les choix (listes déroulantes, champs numériques bornés), on doit valider côté serveur **chaque paramètre reçu**. Utilisez les **contraintes Symfony Validator** sur les DTO ou entités : par ex., dans l'entité Reservation, `@Assert\Positive(message="Le nombre de places doit être au moins 1")` sur `nombrePlaces`. Vous pouvez aussi mettre `@Assert\LessThanOrEqual(value=..., message="...")` dynamiquement, mais la valeur max dépend de la séance. Il est souvent plus simple de faire cette vérification dans le service : `if ($nbPlaces > $seance->getPlacesDisponibles()) erreur`. Vérifiez aussi que l'ID de séance fourni correspond bien à une séance existante en base ; si le formulaire est altéré (ex. quelqu'un essaie de réserver un ID qui n'existe pas ou qui ne correspond pas au cinéma/film attendu), le contrôleur/service doit détecter l'absence ou l'incohérence et retourner une erreur 404 ou 400. Ne faites jamais confiance aux seuls champs hidden ou selects – toujours valider côté back.
  De plus, assurez-vous de la **cohérence métier** : par exemple, empêcher une réservation sur une séance passée (ne pas permettre de réserver hier ou plus tôt dans la journée). Si le service reçoit une date passée, il jette une exception. Pareil, interdiction de réserver 0 place ou un nombre déraisonnable (on peut limiter à une certaine valeur max par réservation si souhaité, ex. 10 places max par commande pour éviter qu'un user ne bloque toute la salle). Ces règles doivent être documentées et implémentées.
- **Validation des données (Front-end)** : En complément, utilisez HTML5 (attributs `required`, `min`, `max` sur le champ nombrePlaces) et Stimulus (comme décrit) pour fournir une validation instantanée. Cela évite au maximum les allers-retours pour des erreurs simples et améliore la sécurité en réduisant les entrées erronées. Cependant, rappelez-vous que cette validation front peut être contournée, d'où la nécessité du back-end robuste ci-dessus.
- **Autorisations (Authorization)** : Dans ce contexte, seules des fonctionnalités utilisateur standard sont envisagées. Mais par précaution, assurez-vous que **seul le propriétaire** d'une réservation peut la voir dans son espace. Si vous listez les réservations de l'utilisateur connecté via le repository filtré par user, c'est bon. Évitez absolument que l'utilisateur puisse consulter la réservation de quelqu'un d'autre (ex: ne pas permettre d'accéder à `/reservation/confirmation/{id}` avec un ID d'une réservation d'autrui). Utilisez l'identifiant de l'utilisateur en session pour filtrer toute donnée sensible. Pareil, lorsque l'utilisateur soumet la réservation, le `userId` ne doit pas être lu du formulaire (on prend `->getUser()` du token de sécurité côté serveur).

Si plus tard il existe un rôle admin pour gérer toutes les réservations, ces interfaces seront distinctes et protégées par ROLE_ADMIN, etc., mais hors du périmètre actuel.

- **Protection contre les injections et XSS** : En utilisant Doctrine et Twig, on est en grande partie protégé des injections SQL (requêtes préparées) et des injections XSS (Twig échappe automatiquement les variables affichées). Continuez à suivre ces bonnes pratiques : ne jamais concaténer de chaînes dans des requêtes SQL manuellement, et échapper toute donnée utilisateur affichée. Par exemple, le nom du film ou du cinéma provenant de la DB est affiché tel quel via `{{ cinema.name }}` — c'est très bien (toute balise potentielle serait échappée). Idem pour d'autres champs. Si vous intégrez du HTML personnalisé (peu probable ici), passez par des whitelists ou fonctions d'assainissement.
- **Logs et suivi** : Implémentez éventuellement un logging des actions de réservation (ex: en cas d'échec de réservation pour place insuffisante, logguez l'événement, cela peut aider à repérer des tentatives anormales). Ce n'est pas explicitement demandé, mais fait partie d'une sécurisation avancée (détection de problèmes).

En appliquant ces mesures, on s'assure que la nouvelle fonctionnalité est **sécurisée et conforme** aux standards. L'utilisateur ne pourra réserver qu'en étant connecté (protection de l'accès), ne pourra pas soumettre de données mal formées ou malveillantes sans se faire bloquer (validation + CSRF), et les données en base resteront intègres (contrôles sur les règles métier, transactions). Ces considérations de sécurité sont essentielles pour une application en production et répondent aussi aux exigences du titre professionnel (développer une application *sécurisée* en couches).

En conclusion, le plan d'action ci-dessus couvre toutes les étapes pour supprimer l'ancienne fonctionnalité de réservation et la reconstruire proprement selon le cahier des charges *Cinéphoria* (US4). En synthèse : on purge le legacy, on redéfinit une base solide (données, architecture), on déroule le parcours utilisateur complet (cinéma → film → séance → réservation) avec une interface moderne (Tailwind) et interactive (Stimulus), tout en respectant les principes de conception (couches Symfony bien séparées, logique réutilisable en services) et en garantissant la qualité (tests unitaires, E2E) et la sécurité (authentification, CSRF, validation). Chaque étape a été détaillée de manière **exhaustive, claire et actionnable** afin que vous puissiez implémenter immédiatement cette refonte. En suivant ce plan, l'utilisateur final disposera d'un module de réservation fiable, maintenable et conforme aux exigences initiales, sans ajout de fonctionnalités non souhaitées mais avec une exécution techniquement améliorée. Bonne reconstruction !