# ***Documentation Technique de l'Application***

## ***I/ Architecture Logicielle de l'Application***

### 1.1/ Pourquoi ces technologies ?

- **Symfony** : Framework PHP robuste pour la gestion du backend et des API, idéal pour une application nécessitant une sécurité renforcée (authentification, gestion des utilisateurs) et des interactions avec une base relationnelle (MySQL) et NoSQL (MongoDB).
- **Flutter** : Framework cross-platform performant, permettant de développer une application mobile riche, tout en réduisant les coûts de développement (ios et android avec le même code).
- **Twig avec Tailwind CSS et JavaScript** : Utilisé pour le rendu des templates sur la partie web, offrant une intégration fluide avec Symfony et un design moderne et responsive grâce à Tailwind.
- **Electron** : Choisi pour développer une application bureautique, permettant un accès direct au backend Symfony pour la gestion administrative sans dépendance d'un navigateur web.

### 1.2/ Le Fonctionnement Global

- **Backend Symfony** : Gère les API REST, l'authentification, API Platform, et la logique métier. Les données sont stockées dans une base MySQL pour la gestion classique (base de données relationnelle) et MongoDB pour les statistiques.
- **Frontend Mobile (Flutter)** : Permet aux utilisateurs d'accéder aux fonctionnalités de réservations (liste des réservations et détailles des réservations) avec une expérience native.
- **Frontend Web (Twig)** : Fournit une interface utilisateur web accessible pour des fonctions clients, employés et administrateurs, optimisée pour la rapidité et la responsivité.
- **Application Bureautique (Electron)** : Offrant une interface pour les employés avec des fonctionnalités avancées comme le listing et la création des problèmes techniques dans les salles.

---

## ***II/ Réflexions Initiales Technologiques sur le sujet***

- **Sécurité** : Mise en place d'une authentification sécurisée (Symfony) et d'un tunnel Cloudflare pour sécuriser les connexions.

- **Accessibilité** : Cross-platform grâce à Flutter et Electron.
- **Scalabilité** : Utilisation de MongoDB pour gérer de grandes quantités de données statistiques sans impacter les performances.
- **CI/CD** : Pipeline DevOps pour déployer rapidement des mises à jour sur les plateformes web, mobile et bureautique.

---

## ***III/ Configuration de l'Environnement de Travail***

### 3.1/ Backend

- Installation de Symfony 7.
- Configuration de Docker Compose pour le backend (PHP, MySQL, phpMyAdmin).
- Tunnel Cloudflare pour rendre le backend accessible localement.

### 3.2/ Frontend Mobile

- Installation de Flutter et configuration avec Android Studio.
- Tests sur simulateurs et appareils physiques.

### 3.3/ Frontend Web

- Intégration de Twig et Tailwind CSS via npm.
- Utilisation de Webpack pour optimiser le chargement des assets.

### 3.4/ Application Bureautique

- Configuration d'Electron avec Node.js.
- Intégration des appels aux API Symfony.

---

## ***IV/ Explication du transaction.sql***

### *4.1/ Objectif de la transaction SQL :*

L'objectif de cette transaction est d'assurer l'intégrité et la cohérence des données lors d'une réservation de sièges pour une séance de cinéma. Cette transaction suit le principe de l'atomicité : soit toutes les étapes sont validées et enregistrées dans la base de données, soit, en cas d'erreur, aucun changement n'est appliqué grâce à un mécanisme de rollback.

Cette approche permet d'éviter des problèmes tels que la double réservation des mêmes sièges ou l'enregistrement d'une réservation incomplète.

***4.2/ Détail de la transaction SQL :***

```
START TRANSACTION;
```

Cette commande indique le début de notre transaction. Toutes les opérations qui suivent seront exécutées en bloc et validées qu'à la fin.

```
SELECT reserved_seats FROM session WHERE id = 1 FOR UPDATE;
```

Cette requête récupère la liste des sièges déjà réservés pour la session spécifiée.
FOR UPDATE permet de verrouiller cette ligne dans la base de données, ce qui empêche tout autre transaction de la modifier tant que celle-ci n'est pas terminée.

```
INSERT INTO reservation (seats, total_price, session_id, user_id, qr_code_url, created_at)
VALUES ( seats '["79", "81"]', total_price 20.00, session_id 1, user_id 1, qr_code_url NULL, created_at NOW());
```

Cette requête insère une nouvelle réservation avec les sièges demandés (79 et 81).
La réservation est associée à un user (user_id = 1) et à une session (session_id = 1).
NOW() permet de mettre la date et l'heure actuelle dans la base.

```
SET @reservation_id = LAST_INSERT_ID();
```

LAST_INSERT_ID() permet de récupérer l'identifiant de la réservation que l'on vient d'insérer.

```
UPDATE session
SET reserved_seats = ('["79", "81"]')
WHERE id = 1;
```

On met à jour la colonne reserved_seats pour inclure les sièges nouvellement réservés.
('['79','81']') permet de stocker ces valeurs sous forme d'un tableau JSON.
On s'assure que les sièges sont bien bloqués pour éviter qu'ils ne soient réservés à nouveau par un autre utilisateur.

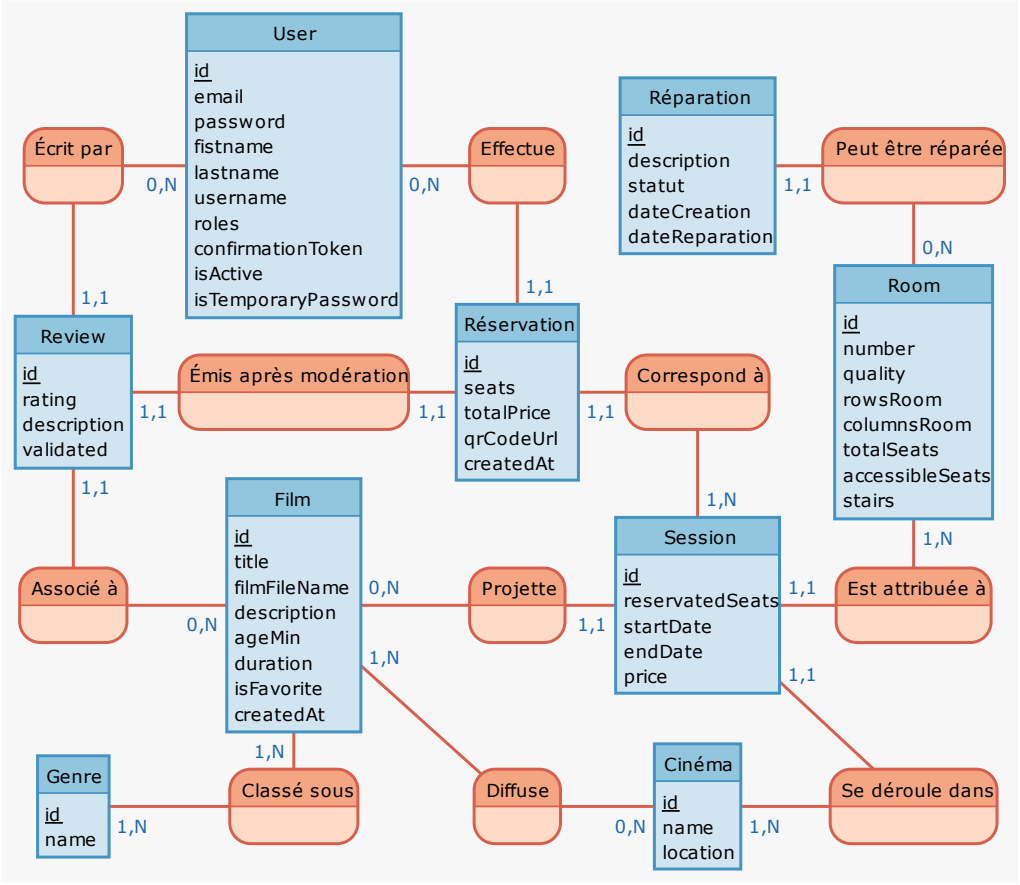
```
COMMIT;
```

Si toutes les opérations précédentes ont réussi sans erreur, cette commande applique définitivement les modifications dans la base de données.
À partir de ce moment, la réservation est considérée comme confirmée.

```
-- ROLLBACK;
```

Si une erreur survient avant que COMMIT ne soit exécuté, on peut exécuter ROLLBACK pour annuler toutes les modifications effectuées durant la transaction.

## V/ Modèle Conceptuel de Données (MCD)

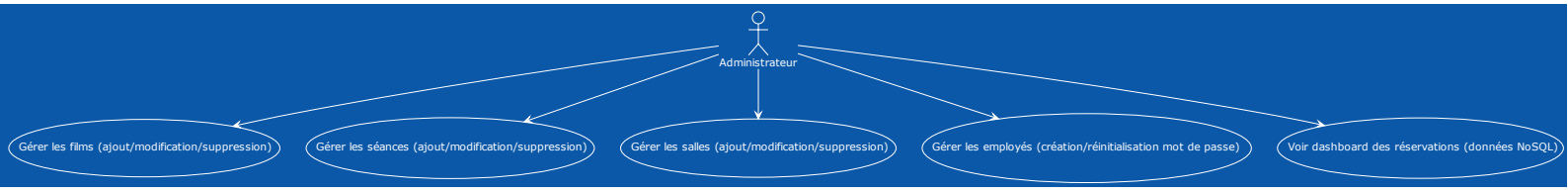



---

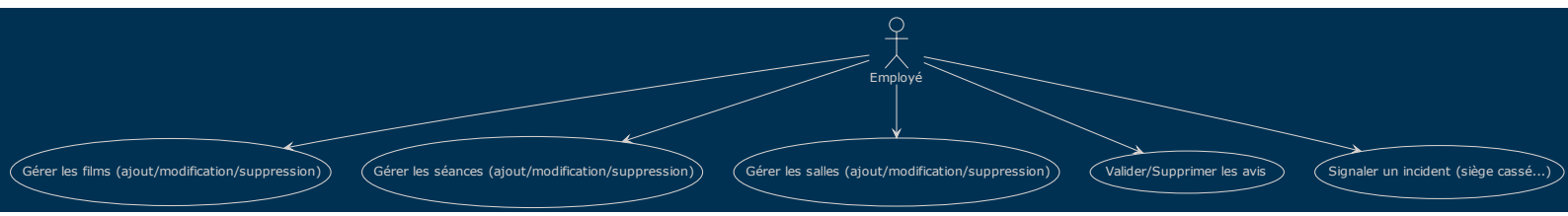
## VI. Diagramme d'Utilisation et de Séquence

### *6.1 Diagramme d'Utilisation*

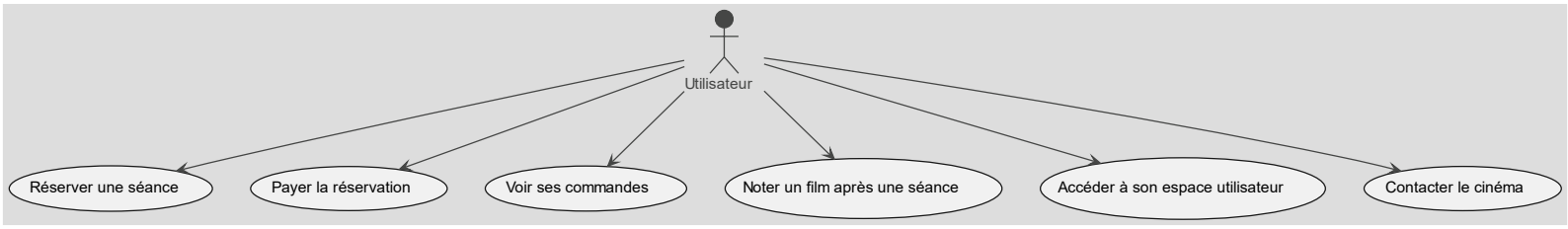
#### *6.1.1/ Diagramme de cas d'utilisation Administrateur*



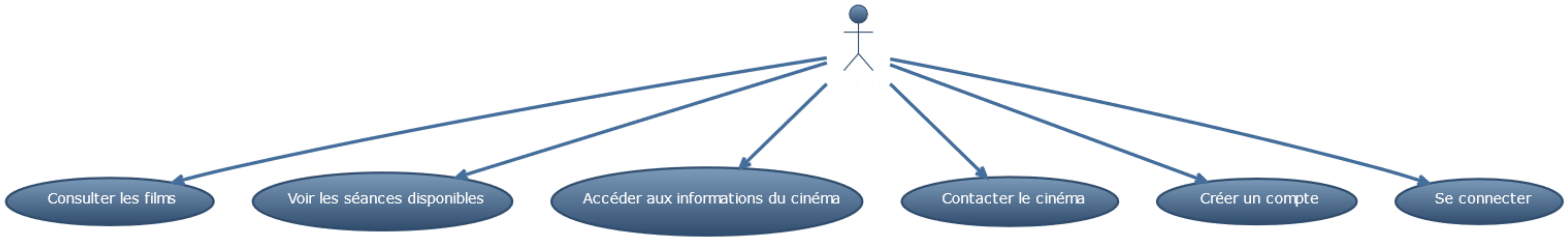

#### 6.1.2/ Diagramme de cas d'utilisation Employé



### 6.1.3/ Diagramme de cas d'utilisation Utilisateur

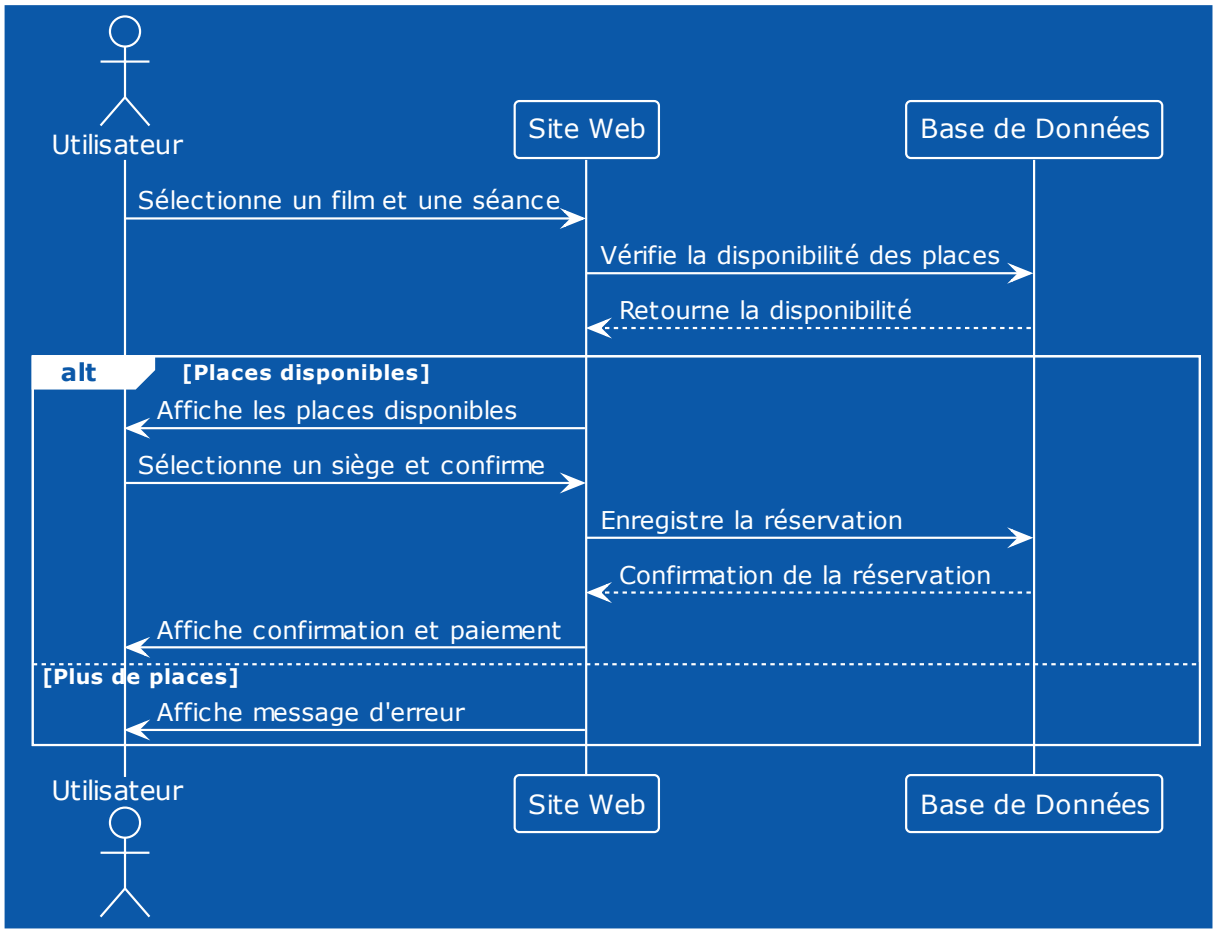



### 6.1.4/ Diagramme de cas d'utilisation Visiteur




## 6.2/ Diagramme de Séquence

Exemple : Processus de Réservation



## VII/ Plan de Test

### 7.1/ Test Fonctionnel

J'ai réalisé un test fonctionnel (disponible dans le github). Il s'appuie sur WebTestCase, qui permet de simuler des requêtes HTTP et de vérifier si les endpoints fonctionnent correctement. L'objectif principal est de s'assurer que les fonctionnalités de réservation fonctionnent correctement.

Le test commence par importer plusieurs entités comme Cinema, Film, Genre, Room, et Session. Dans la méthode setUp(), le client HTTP est initialisé avec static::createClient(), et l'EntityManager est récupéré afin de manipuler la base de données directement pendant le test.

Le premier test, testIndex(), vise à vérifier que l'endpoint « /reservation » fonctionne correctement lorsqu'un paramètre cinema est fourni. Pour cela, le controller crée un cinéma, un genre, une salle, un film et une séance, qu'il persiste en base de données. Ensuite, il envoie une requête GET à l'URL « /reservation » et s'assure que la réponse est bien un succès grâce à assertResponseIsSuccessful().

Le second test, testConfirmReservation(), teste le processus de confirmation d'une réservation. Il commence par créer un utilisateur et d'autres entités nécessaires, comme un cinéma, un film et une séance. Une fois ces éléments en place, le test simule la connexion de l'utilisateur avant d'envoyer une requête de réservation. Il vérifie ensuite que la réponse reçue est correcte et que la réservation a bien été prise en compte.

Enfin le dernier test simule l'affichage d'une réservation existante. Après avoir créé un cinéma, une salle, un film, une session et un utilisateur, une réservation est enregistrée en base avec des sièges réservés et un QR code. Une requête GET est envoyée sur /reservation/view/{id}, et le test s'assure que la réponse est bien 200 OK.

```
...                                                                 3 / 3 (100%)

Time: 00:00.925, Memory: 42.00 MB

OK (3 tests, 3 assertions)
```

On observe ici que le test fonctionnel a correctement réussi.

---

## VIII. Déploiement Continu (CI/CD)

Le déploiement continu (CI/CD) d'un site Symfony sur un VPS OVH implique plusieurs étapes pour automatiser le déploiement de mon site Cinéphoria. Voici les étapes générales de la configuration d'un pipeline CI/CD à l'aide de GitHub Actions, avec un déploiement sur mon VPS.

### *8.1/ Préparation de mon VPS OVH*

- Installation d'un serveur web (Apache)
- Installation de PHP
- Installation de Composer
- Installation de Symfony
- Installation de Git pour pouvoir cloner mon dépôt.
- Configuration d'une clé SSH.

### *8.2/ Configurer le dépôt GitHub*

- Mon projet est dans un répo GITHUB
- J'ai créer un fichier de workflow GitHub Actions (.github/workflows/deploy.yml) pour définir le pipeline CI/CD.

### *8.3/ Configurer les secrets GitHub*

- J'ai ajouté ma clé privée SSH et mes informations du .env dans les secrets de mon dépôt GitHub (le .env se génére dans les actions).

### *8.4/ Configurer le VPS OVH*

- J'ai configure mon serveur web pour pointer vers le répertoire public/ de mon projet Symfony.
- Enfin, j'ai verifié que les permissions des fichiers et répertoires sont correctes.