Cinéphoria

Documentation technique

Jérémy Sananikone

21 novembre 2024

Architecture logicielle de l'application

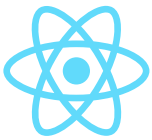
php 8

nextjs

typescript

expo

symfony 7.1



react/reactnative

Cinéphoria

api platform

vitejs

doctrine orm

electronjs

docker

Mysql 8.0

mongodb

Réflexions initiales sur le sujet

Selon moi, le défi reposait sur deux aspects. D'une part, l'ampleur du projet. C'est à dire la nécessité de créer trois applications en 4 mois. D'autres part, la prise en compte de la notion du temps, déterminante dans la bonne l'application des règles métiers. En effet, le sujet met en scène des séances de cinéma à vendre. Par définition, une séance est par sa durée de vie. Cela engendre en conséquence des contraintes métier à implémenter, telles que l'invalidation de certaines actions en fonction de la date courante.
Ces contraintes amènent à envisager des solutions technologiques telles que des crons ou scheluder (propre à symfony).

Concernant le back-end, il m'est apparu évident de construire une api qui serait consommée par les autres applications. De cette manière, tout le coeur de la logique métier est concentrée sur une seule application.

Explication du choix des technologies

J'ai donc utilisé :

- Symfony avec Api platform pour l'api
- Next JS pour le front-end
- React Native avec expo pour l'application mobile
- Electron JS pour l'application Desktop.
- Docker pour le déploiement.
- Mysql pour la base de données

Symfony couplé avec Api platform permet de mettre en place rapidement une api sécurisée. Le travaille se concentre principalement sur la configuration des entités avec l'usage d'attributs php. La logique métier s'implémente dans les providers et les processors, ainsi que dans les contrôleurs (bien que déconseillé par la documentation d'api platform). La sécurité et la rapidité de mise en place permise par ces technologies m'ont conforté dans ce choix, bien qu'au début j'envisageais Nest JS pour une approche plus modulaire du code. Concernant l'application web, j'ai choisi Next JS pour le rendu côté serveur, ce qui constitue un avantage pour le SEO par rapport à une application React traditionnelle. Afin de rester dans un environnement React, j'ai naturellement choisi React Native pour l'application mobile. Pour l'application desktop, j'ai là aussi utilisé React avec Vite au sein du framework Electron JS. Le choix de Docker provient de mon souhait de ne dépendre d'aucune plateforme de déploiement telle que Heroku ou Fly.io.
Ces choix techniques me sont apparus adaptés à la demande initiale.

Concernant la gestion de projet

Au commencement, je voulais appliquer la méthode SCRUM avec la construction d'un projet Jira appuyé par un projet Confluence (lien du confluence). Je voulais mettre en place un product backlog organisé en tickets, eux-mêmes distribués sur différents sprints. Ayant travaillé seul sur le projet, je n'ai malheureusement pas réussi à appliquer une telle rigueur, et me suis finalement résigné à l'usage du tableau Kanban seul.

Le tableau Kanban fait parti des méthodes agiles reconnue et utilisées. Il s'applique facilement à toute taille de projet, seul ou en équipe. C'est la raison pour laquelle je l'ai choisi, au détriment d'une organisation SCRUM qui prend tout son sens en équipe.

J'ai donc construis un tableau trello contenant une liste des fonctionnalités à implémenter. Dans un premier temps, je note toutes les idées me traversant l'esprit, à la manière d'un brainstorming. J'essaie ensuite de trier les étiquettes crées par catégorie (conception, back-end, front-end, desktop, mobile).

Cependant, il est important de savoir que certaines idées d'implémentation me sont apparus pendant l'écriture du code, et pendant mon temps de recherche diverses documentations techniques. Ce tableau Kanban a donc évolué en même temps que le projet.

Configuration de l'environnement de travail

Mise en place de git

Cela commence par l'initialisation des projets dans leur language respectif et des dépôts git distants. De manière générale, je commence par travailler sur la branche principale (main). Arrivé à un certain stade du développement, je commence à implémenter les nouvelles fonctionnalités dans des branches distinctes. Une fois satisfait, je merge les branches de développement dans la principale avec une pull request.

Bien que l'idée ne m'est venue que plus tard dans le développement, j'ai mis en place un dépôt parent distant dont chaque application (api, front-end, mobile, desktop) est un submodule. Cela permet de les regrouper dans un même dépôt tout en conservant l'historique de chacun.

Mise en place du projet symfony et des bases de données

J'utilise les images docker officielles de mysql et mongodb que j'execute dans des conteneurs en local. L'application symfony est lancé avec le serveur php via la commande **symfony server:start** et communique avec ces conteneurs**.** Si j'ai besoin de consulter le contenu de la base de donnée, je me connecte au conteneur mysql dans lequel j'utilise le client mysql en ligne de commande. Pour la noSQL, j'utilise Mongodb Compass.

Mise en place du projet web

J'ai suivi la documentation officielle de NextJS pour initialiser le projet next avec typescript. Dans le fichier next.config.mjs, j'ai rajouté une configuration remotePatterns pour faire en sorte que les images soient acceptées depuis

n'importe quel hôte en http comme en https. Pour des raisons de sécurité, il sera envisagé plus tard de réduire cette permission.

Mise en place du projet mobile

J'ai suivi la documentation officielle d'Expo pour initialiser le projet avec typescript.

J'initialise une variable d'environnement **EXPO_PUBLIC_API_URL** dans laquelle je renseigne l'url de l'api déployée. Pour utiliser l'application mobile avec l'api fonctionnant en local, il faut utiliser l'utilitaire ngrock, ou tout simplement remplacer localhost par l'addresse ip de la machine hôte dans l'url atribuée à la variable **EXPO_PUBLIC_API_URL**.

Mise en place du projet desktop

Pour ce projet, j'ai voulu utiliser react avec vite et typescript dans l'environnement d'electronJS. Electron est un framework permettant de construire des applications de bureautique dans un environnement web. Il est construit sur chromium, un navigateur web open source. Pour initialiser le projet, j'ai suivi les trente première minutes de cette vidéo : https://www.youtube.com/watch?v=fP-371MN0Ck.

Cela commence par l'initialisation d'un projet react avec Vite js. Il s'agit d'un bundler javascript préconisé par la documentation officielle de react. Je configure le dossier de build sur « dist-react » dans le fichier vite.config.ts. Ensuite j'installe electron avec la commande *npm install –save-dev electron*.

Dans le dossier /src, je sépare le code react du code electron en mettant le premier dans un dossier /ui, l'autre dans un dossier /electron. Comme pour le dossier dist-react, je crée un dossier de build /dist-electron pour le code concernant electron.

Enfin, je configure les commandes exécutables dans le fichier package.json, de manière à lancer en une commande *npm run dev* :

- le compilateur typescript pour le code react et le code electron
- le serveur vite
- le serveur electron

Conception avec la méthode Merise (MCD)

Les documents sont disponibles dans le dossier /livrables/Merise au format .pdf.

Diagramme d'utilisation, diagramme de séquence

Les documents sont disponibles dans le dossier /livrables/Diagrams au format .pdf.

Charte graphique, wireframe et mockups

Les documents sont disponibles dans les dossiers suivant au format .pdf :

- /livrables/Wireframes
- /livrables/mockups
- /livrables/graphic_charter.pdf

Les tests

Décrivez les tests applicatifs que vous effectués et expliquez dans quel but.

Actuellement, seule l'application Symfony dispose de tests. J'ai choisi de tester l'API car elle joue un rôle centrale où est concentrée la logique métier. D'autres tests sont en cours d'écriture mais ils seront écris après la date de rendu.

J'ai voulu tester la fonctionnalité principale de l'application : l'achat de billets pour une séance de cinéma. Cette mécanique doit être tester en plusieurs étapes :

- l'appel au endpoint «/checkout ».
- l'appel au webhook «/payment-webhook » par le service stripe

J'ai donc écris un test pour la route «/checkout» de la classe CheckoutController (celui pour le webhook est en cours d'écriture). Cette route est appelée lorsque l'utilisateur choisi et valide le tarif de ses billets sur l'application front-end. Il est ensuite redirigé vers la page checkout de Stripe. Il s'agit donc d'une approche fonctionnelle dans la mesure où il s'agit de tester le comportement d'un contrôleur qui fait appel à différents services et à la base de données.

L'idée n'est donc pas de mocker les données, mais plutôt de servir au contrôleur de vraies données de test. Il était donc nécessaire de mettre en place une base de données de test. Pour cela j'ai utilisé phpunit avec les bundles symfony/test-pack et liip/test-fixtures-bundle. Ce dernier permet de charger des fixtures avant le lancement des tests. C'est à dire qu'il recrée la base de données de test avec les tables, et insert les données avant de faire des assertions.

Dans le cadre du test **testCheckoutController()**, Je m'assure d'abord d'insérer en base, plusieurs réservations dont la date de création est inférieure ou supérieur à cinq minutes par rapport à l'heure actuelle. Ensuite, le test effectue

pour chaque réservations une requête POST sur la route /checkout. Pour les réservations n'ayant pas dépassé cinq minutes, le test vérifie que la réponse HTTP contienne l'url de redirection vers la page checkout de Stripe. Pour les autres, il vérifie que le code HTTP renvoyé est 410 (« Gone », sous-entendu que la réservation n'existe plus, car elle a été supprimée par le scheduler de Symfony).

Le déploiement continue

L'application front et l'api sont déployées. La première étant une application Next JS, j'ai utilisé la solution de déploiement continue proposée par Vercel. Tout changement sur la branche main est aussitôt déployé en production.

Concernant l'api, cela commence par la création du Dockerfile à la racine du projet Symfony :

Je pars de l'image docker officielle php:8.2-apache**.** Ensuite, j'installe les packages linux nécessaires au fonctionnement du projet avec la commande apt-install (voir Dockerfile pour le détail). Je copie ensuite tout le contenu à la racine du projet dans le dossier « /var/www/html » de l'image (excepté ce qui est renseigné dans le .dockerignore, à savoir les dossiers vendor, node_modules, .env et d'autres…). Je termine par l'execution d'un script shell qui s'effectue à chaque démarrage du conteneur (*voir le fichier /shell/start.sh*). Ce script fait plusieurs choses dont :

- l'initialisation de la base de données
- L'optimisation du cache en mode production
- La génération d'une nouvelle paire de clés publique/privée pour la génération des JWT.
- Le lancement des schedulers de symfony.
- Le lancement de apache.

Une fois le Dockerfile crée, j'ai mis en place un pipeline de déploiement continue avec Github actions. Cela passe par la création d'un fichier .github/workflows/deploy.yaml dans lequel deux jobs sont définis. Le premier s'occupe de construire l'image et l'envoie ensuite sur le docker hub (seulement pour l'architecture AMD afin de pouvoir l'utiliser sur mon VPS. Pour une image sur architecture ARM, il faudra la construire directement une machine ARM). Le second job est chargé de de connecter en ssh au VPS pour télécharger l'image et lancer docker compose.

Le SQL

Des scripts SQL sont disponibles dans le dossier /livrables/sql_scripts du dépôt parent. Vous les trouverez aussi dans le dossier /sql du depôt du submodule api.

Il y a parmi ces fichier:

- un script de création de la base de donnée : **/0_initdb.sql**
- un script d'insertion des données liées aux utilisateurs : **/1_insert_users.sql**
- un script d'insertion des données liées aux cinémas, salle de projection, sièges, les catégories de film, les tarifications des billets, les qualités de projection : **2_insert_data.sql**
- Un export de base de donnée avec mysqldump : **/export_mysqlDump.sql**
- Un script décrivant une transaction : **/transaction.sql**

Explication de la transaction sql

Ce script intervient une fois que l'utilisateur a soumis une demande de paiement sur la page de checkout de Stripe. Une fois la demande effectué, le service Stripe appel le webhook sur la route /payment-webhook. Ce contrôleur est chargé de mettre à jour le statut de la réservation concernée et de générer les tickets achetés. Ces opérations s'effectue dans l'appel au script / **transaction.sql**. Cet appel s'effectue grâce à l'objet entityManager de Doctrine qui injecte deux paramètres au script : l'identifiant de la réservation, et un tableau contenant des chaînes de caractères. Chaque string correspond à une catégorie de billet et il y a autant d'items dans le tableau qu'il y a de billet à générer.

Le script SQL démarre la transaction avec l'instruction BEGIN TRANSACTION. La première opération change la propriété is_paid de la réservation à true. La seconde opération génère les tickets. Pour cela, j'utilise une table éphémère générée à la volée, communément appelée CTE[1] :

[1] CTE: Common Table Expressions

```
WITH RECURSIVE ticket_numbers AS (
    SELECT
        1 AS ticket_number,
        :reservation_id AS reservation_id,
        (SELECT id FROM ticket_category WHERE category_name
= JSON_EXTRACT(:tickets, CONCAT('$[', 0, ']'))) AS
category_id,
        UUID() AS unique_code,
        @current_date AS created_at,
        @current_date AS updated_at,
        FALSE AS is_scanned
    UNION ALL
    SELECT
        ticket_number + 1,
        :reservation_id as reservation_id,
        (SELECT id FROM ticket_category WHERE category_name
= JSON_EXTRACT(:tickets, CONCAT('$[', ticket_number, ']')))
AS category_id,
        UUID(),
        @current_date,
        @current_date,
        FALSE
    FROM ticket_numbers
    WHERE ticket_number < JSON_LENGTH(:tickets)
)
```

Cette table contient les même colonnes que la table ticket originale. Les CTE sont utilisée pour générer des données de manière récursive. L'idée est donc de générer autant de lignes que d'items contenus dans le tableau injecté en paramètre du script. Pour faire cela, il y a d'abord une instruction SELECT qui va initialiser la première ligne du tableau, et à partir de laquelle seront générées les suivantes. In compteur est généré sur cette ligne :

```
1 AS ticket_number,
```

Les lignes suivantes sont générées à partir de l'instruction UNION ALL. La fonction UUID() de mysql permet de générer des codes uniques.

Finalement, le contenu de cette CTE est injecté dans la table des tickets. La transaction est ensuite terminée avec l'instruction COMMIT. Si une erreur était arrivé, un rollback aurait annulé toutes les opérations effectuées.

La sécurité

Plusieurs procédés ont été utilisés pour sécuriser les applications :

- L' usage de requêtes SQL préparées. Doctrine ORM fait automatiquement ce travaille.
- Les mots de passe sont hashés avec l'algorithme bcrypt utilisé par Symfony.
- L'usage de tokens csrf dans les formulaires rendus avec le moteur de template Twig. Cela permet de s'assurer qu'un formulaire soumis a bien été généré par le serveur lui-même.
- l'usage de fonctions natives au langage utilisé : htmlspecialchars() et filter_var() en php.
- L'utilisation d'un certificat TSL lié à mon nom de domaine. J'ai utilisé pour cela l'utilitaire certbot qui permet de générer un certificat auprès de l'organisme Let's Encrypt. Ensuite toutes les requêtes en http sont redirigé en https avec Nginx qui jour le rôle de reverse proxy.
- Je prévois aussi l'utilisation des CSP pour limiter d'avantage les failles XSS (ce n'est pas encore implémenté dans le code, mais cela se fait dans la configuration de Nelmio).
- Le changement du numéro de port de connexion ssh à mon vps (port 20 par défaut).
- Bien qu'il s'agisse d'une sécurité contournable, l'usage de contraintes sur les formulaires de l'application web (avec zod en react) permet d'empêcher l'utilisateur de saisir des données erronées.