PROJET DE FIN DE FORMATION

Lucas Lefèvre
Stagiaire Développeur Web Full Stack
Formation GRETA - Kercode 2024/2025

Lien projet sur Github : https://github.com/lucaslefevre56/cine-hurlant

CINÉ-HURLANT

ANALYSES CROISÉES ENTRE CINÉMA ET BD DE SCIENCE-FICTION

1. Introduction personnelle

Avant d'intégrer la formation Kercode, j'étais ouvrier, sans bagage dans le numérique. Je suis arrivé avec l'envie de me reconvertir dans un domaine plus créatif, stimulant et porteur d'avenir. J'ai découvert le développement web comme une manière de concilier logique, esthétique, narration et liberté technique.

Ce projet de fin d'année est une synthèse de ce que j'ai appris pendant 7 mois. Il illustre non seulement mes compétences techniques, mais aussi mon approche autonome, mon intérêt pour la culture geek et ma capacité à structurer un site complexe en pensant à la sécurité, l'expérience utilisateur et la maintenabilité.

2. Présentation du projet

Le site **Ciné-Hurlant** explore les liens artistiques et thématiques entre la bande dessinée française Métal Hurlant et les grands films de science-fiction. Il propose des analyses croisées, une base d'œuvres, un moteur de recherche, un système de commentaires, et un espace de gestion pour les rédacteurs et administrateurs.

Pourquoi ce sujet ?
Parce que c'est une manière de relier mes passions (le cinéma, la BD, la SF) à un projet technique réel. Ce site n'est pas un simple blog : il structure des données, propose de la recherche intelligente, gère plusieurs types de rôles utilisateurs, et repose sur une architecture MVC entièrement codée à la main sans framework.

Objectifs principaux :

- Proposer un contenu original, illustré et interactif
- Permettre une contribution dynamique par les rédacteurs
- Gérer les commentaires sans rechargement de page (AJAX)
- Sécuriser l'ensemble des actions utilisateur
- Rendre le tout accessible, responsive et optimisé pour le référencement

3. Architecture MVC expliquée

J'ai choisi une architecture **MVC (Modèle-Vue-Contrôleur)** pour plusieurs raisons :

- **Organisation claire** : je sépare la logique (contrôleur), les données (modèle) et l'affichage (vue).
- **Maintenance facilitée** : chaque fichier a un rôle précis, ce qui permet de déboguer facilement.
- **Bonne pratique professionnelle** : même sans framework, j'ai pu mettre en place une architecture proche de ce qui se fait dans le monde pro.

Exemples de structure :

- Les routes sont analysées dans Router.php (pas de framework)
- Les modèles comme Oeuvre.php, Article.php ou Utilisateur.php héritent de BaseModel.php pour centraliser la sécurité SQL
- Chaque contrôleur traite les entrées, appelle les bons modèles, et transmet les résultats aux vues

Vue AJAX vs Vue classique

J'ai aussi distingué les vues classiques et les vues partielles pour le chargement dynamique. Par exemple, dans l'admin ou le panel rédacteur, les onglets sont chargés via fetch(), et j'utilise renderPartial() pour éviter de recharger tout le layout.

4. Technologies utilisées

Côté Front-End :

- **HTML5 / CSS3** : Structure des pages, accessibilité, animations simples.
- **SASS / SCSS** (Mobile First) : j'ai adopté une stratégie de responsive design adaptée au mobile en premier, avec des styles modulaires et organisés.
- **JavaScript** : interactivité (AJAX, carrousels, recherche dynamique, modales...)

Côté Back-End :

- **PHP natif** : tout est fait à la main, sans Laravel ou autre framework.
- **PDO** : pour les connexions SQL sécurisées.
- **MySQL** : pour la base de données relationnelle (gestion des utilisateurs, articles, œuvres, genres, etc.)

Outils de travail :

- **VS Code** : environnement de développement
- **phpMyAdmin** : pour la gestion de la BDD
- **Looping** : pour modéliser la base
- **Figma / Canva** : pour les maquettes graphiques
- **Git + GitHub** : versioning local et en ligne

5. Logique fonctionnelle et gestion des rôles

Une plateforme pensée par niveaux d'accès

Dans mon projet Ciné-Hurlant, j'ai structuré la plateforme autour de **rôles utilisateurs bien distincts**, pour répondre à la fois aux besoins de consultation, de création de contenu, et d'administration du site. Cette logique reflète la gestion réelle d'une communauté éditoriale en ligne.

J'ai donc défini 4 rôles :

- **Visiteur** : peut consulter les œuvres, lire les articles et voir les commentaires.

- **Utilisateur enregistré** : peut commenter les œuvres et les articles.

- **Rédacteur** : peut ajouter, modifier et supprimer ses propres articles et œuvres via un panneau personnel.

- **Administrateur** : peut tout gérer (contenus, utilisateurs, rôles, commentaires) via un back-office dédié.

Chaque niveau d'accès déclenche des **affichages conditionnels dans l'interface** et des **vérifications de rôle côté serveur**, pour éviter les failles d'autorisation. J'ai encapsulé cette logique dans un helper AuthHelper.php pour centraliser toutes les fonctions de type isAdmin(), isRedacteur(), isLoggedIn(), etc.

Une navigation fluide et structurée

Pour simplifier la navigation malgré la richesse des contenus, j'ai mis en place :

- Un **menu principal dynamique**, qui change selon le rôle de l'utilisateur connecté.

- Une **page d'accueil** organisée en blocs visuels (slider d'articles récents + suggestions d'œuvres aléatoires).

- Des **sous-onglets** pour classer les œuvres en *films* ou *bandes dessinées*, mémorisés dans le navigateur.

- Un **moteur de recherche** dynamique en AJAX, qui affiche les résultats dès la saisie (avec filtre par type).

Ces éléments ont été conçus pour **éviter les rechargements de page inutiles**, **minimiser les clics** pour accéder à l'information, et offrir une **expérience cohérente** que l'on soit utilisateur, rédacteur ou simple curieux.

Optimisation de l'expérience utilisateur

Mon objectif n'était pas seulement de "faire fonctionner" le site, mais de le rendre **agréable à utiliser** :

- Les **messages de succès / erreur** disparaissent automatiquement après 5 secondes, sauf exceptions critiques.

- Toutes les actions sensibles (suppression, désactivation, annulation) sont **protégées par une modale de confirmation**.

- Les contenus sont **dynamiquement chargés** dans les panels admin/rédacteur via fetch() → sans recharger toute la page.

- L'interface est **mobile-first**, avec une refonte prévue en SCSS pour garantir un rendu propre sur téléphone.

Une logique de rédaction pensée pour les auteurs

Les rédacteurs disposent de leur propre panneau de gestion, distinct de celui de l'administrateur. Cette séparation est importante pour deux raisons :

1. Elle **limite les risques de mauvaise manipulation** (un rédacteur ne peut pas modifier une œuvre ou un article qui ne lui appartient pas).

2. Elle **simplifie l'interface** : un rédacteur voit uniquement ses contenus, classés par type, avec options de modification ou suppression directe.

Toutes les données sont filtrées dès le contrôleur (getByAuteur()), ce qui garantit une logique de sécurité forte sans alourdir la vue.

Séparation claire des responsabilités

Chaque partie du site a été pensée pour **avoir une mission claire** :

- Les **modèles** interagissent uniquement avec la base de données.

- Les **contrôleurs** gèrent la logique métier (pagination, filtrage, redirection).

- Les **vues** affichent les données sans aucune logique métier.

Ce respect du MVC m'a aidé à organiser le projet de façon lisible et maintenable.

6. Sécurité : un site robuste, même sans framework

Validation côté serveur : une base essentielle

Même si j'ai mis en place des vérifications en JavaScript pour améliorer l'expérience utilisateur, **je n'ai jamais compté uniquement sur le client**. Toutes les données envoyées au serveur sont validées à nouveau : je vérifie qu'elles sont bien présentes, du bon type, et dans le bon format. Cela empêche qu'on puisse contourner les règles en modifiant le code dans le

navigateur.
Chaque formulaire a donc un double contrôle, et le serveur renvoie toujours une réponse claire et sécurisée, sans jamais révéler d'informations sensibles.

Requêtes préparées : une sécurité intégrée dès la base de données

Pour toutes les interactions avec ma base de données, j'ai utilisé des **requêtes préparées** via PDO. C'est la manière la plus sûre d'écrire des requêtes en PHP, car elle empêche les injections SQL.
Au lieu d'écrire des requêtes dynamiques en insérant directement les valeurs, j'ai utilisé des espaces réservés pour que les données de l'utilisateur soient traitées à part et sécurisées automatiquement.
Cela m'a permis d'avoir une base de données protégée contre les attaques classiques, sans avoir besoin d'ajouter des protections manuelles à chaque requête.

Affichage des données : je bloque toutes les failles XSS

Quand un utilisateur écrit un commentaire, un titre ou toute autre information, je m'assure **qu'aucun script ne peut être injecté** dans la page.
Pour cela, j'ai mis en place une **protection systématique à l'affichage**, en échappant tous les caractères spéciaux.
Même les contenus chargés via AJAX sont nettoyés, pour éviter que du JavaScript ne soit exécuté par erreur.
Cette protection est invisible pour l'utilisateur, mais elle empêche les failles les plus dangereuses.

Authentification : des mots de passe protégés et une gestion propre

Tous les mots de passe sont **hashés de manière sécurisée** avant d'être stockés. Cela signifie qu'ils ne sont jamais enregistrés en clair, ni même accessibles dans la base.
Je respecte les bonnes pratiques de sécurité en ajoutant une politique de mot de passe fort (majuscule, minuscule, chiffre, caractère spécial).
Lors de la connexion, je vérifie toujours que l'utilisateur est actif avant de l'accepter, et je gère les rôles pour qu'un utilisateur ne puisse pas accéder à des fonctions réservées à un rédacteur ou un admin.

Gestion des erreurs : propre et sans fuite d'information

En cas de problème, que ce soit une URL incorrecte ou une erreur de base de données, **je ne montre jamais l'erreur technique à l'utilisateur**.

À la place, j'ai mis en place des pages d'erreur claires (404 ou 500), avec un message compréhensible mais sans détail sur l'origine du problème.
Cela permet de sécuriser le site tout en offrant une expérience propre.
De mon côté, je garde la possibilité de suivre les erreurs si besoin, mais sans jamais les exposer dans le navigateur.

Accès aux fichiers sensibles : verrouillé côté serveur

J'ai ajouté dans mon fichier de configuration Apache une **protection spécifique pour les fichiers critiques** : .env, .sql, fichiers de logs…
Même si quelqu'un connaît leur nom, il ne pourra pas y accéder via le navigateur.
C'est une protection essentielle pour éviter toute fuite de configuration ou de données.

Navigation sécurisée : forcer le HTTPS

Pour que toutes les données échangées entre le navigateur et le serveur soient chiffrées, j'ai inclus une **redirection automatique vers HTTPS**.
Cela empêche les interceptions de données, en particulier lors des connexions ou des changements de mot de passe.
Cette mesure est aujourd'hui indispensable pour tout site sérieux, même sans traitement de paiement.

7. Accessibilité, ergonomie et design : un site pour tous

Une navigation claire pensée dès le wireframe

J'ai réfléchi à l'arborescence du site dès les premières maquettes pour **guider l'utilisateur sans qu'il ait à chercher**.
Le menu principal permet d'accéder directement aux zones essentielles : Accueil, Œuvres, Articles, Connexion ou Profil.
Les rôles utilisateurs influent aussi sur le menu (par exemple, un rédacteur verra "Rédacteur Panel", un admin aura "Administration").
C'est une logique simple, mais efficace : je **m'adapte à l'utilisateur** pour alléger son interface et éviter les erreurs.

Responsive mobile-first : je commence par le plus contraignant

Pour garantir que le site soit fluide sur mobile, j'ai adopté une **approche mobile-first** dans le CSS (avec SASS).
Cela veut dire que je pars des petits écrans, puis j'ajoute des ajustements (media queries) pour les tailles supérieures.
C'est plus logique aujourd'hui, car **la majorité des visiteurs** utilisent un smartphone.
J'ai testé le site sur différentes tailles d'écran, et je me suis assuré que les boutons, textes et blocs restent accessibles partout, même sans souris.

Couleurs, typographie : un univers graphique cohérent

Le moodboard que j'ai conçu avec Figma m'a aidé à définir **l'identité visuelle** du projet.
Je me suis inspiré des codes rétro-futuristes de Métal Hurlant : typographies vintage, blocs de couleurs chaudes.
Mais je suis resté lisible : fond clair, texte sombre, tailles confortables.
Les boutons sont bien visibles, et les éléments interactifs (onglets, slider, carrousel) réagissent au survol ou au focus.

Expérience utilisateur : garder la fluidité sans surcharger

Le site est fluide car j'ai évité les rechargements de page dès que possible.
Par exemple :

- Les panels admin et rédacteur sont **chargés dynamiquement via fetch()**
- Les commentaires sont **ajoutés, modifiés ou supprimés sans recharger**
- La recherche est **instantanée pendant la saisie**
- Les messages d'erreur ou de confirmation **s'effacent seuls après quelques secondes**

Tout ça améliore le confort de navigation.
L'idée, c'est qu'on **ne "subisse" jamais le site** : il répond vite, il est clair, il est logique.

8. Gestion des rôles et des contenus : chaque utilisateur à sa place

Quatre niveaux d'utilisateurs, quatre types d'accès

J'ai mis en place un système de rôles clair :

- **Visiteur** → peut lire les œuvres, les articles, et voir les commentaires.
- **Utilisateur inscrit** → peut commenter (avec modification/suppression de ses commentaires).
- **Rédacteur** → peut ajouter, modifier, ou supprimer ses propres œuvres et articles.
- **Administrateur** → a accès à tout : gestion des utilisateurs, validation, suppression de n'importe quel contenu.

Ce système permet de **sécuriser chaque fonctionnalité** et de répartir les responsabilités sans confusion.

Le panneau d'administration : une interface fluide et puissante

L'admin peut gérer tout le site depuis une interface centralisée, sans rechargement.
Il peut :

- Modifier ou supprimer les œuvres, articles, commentaires, comptes
- Attribuer ou retirer un rôle à un utilisateur
- Visualiser en temps réel tous les contenus via des onglets dynamiques

Chaque action affiche une modale de confirmation avant exécution, pour **éviter les suppressions accidentelles**.
L'interface est simple mais efficace, et surtout **100 % en AJAX**.

Le panneau rédacteur : une interface dédiée à la contribution

Le rédacteur a aussi une interface dédiée, plus simple. Il peut :

- Voir tous ses articles et œuvres classés en onglets
- Modifier ou supprimer uniquement ses contenus
- Accéder rapidement à ses créations par type (films ou BD)

Cette séparation me permet de **garder le site organisé**, et de ne jamais afficher à un utilisateur plus que ce qu'il peut gérer.

9. Recherche, pagination : naviguer facilement dans les contenus

La recherche en direct : rapide, intuitive, filtrable

J'ai conçu un moteur de recherche live, qui affiche des résultats pendant que l'utilisateur tape.
L'utilisateur peut choisir de filtrer par type (œuvre, article, ou tous), et les résultats sont :

- Affichés par catégorie (avec lien direct vers la fiche)
- Fermables en un clic
- Mis à jour dynamiquement à chaque frappe

L'objectif était de rendre **la recherche agréable**, sans être trop technique.
Et côté sécurité, j'échappe tous les caractères spéciaux pour **éviter les injections ou XSS**.

La pagination : plus de contenus, mais sans charger trop d'un coup

Pour éviter que les pages deviennent trop longues, j'ai mis en place une **pagination serveur**.
Je calcule le nombre total d'éléments, puis je divise par le nombre d'éléments par page.
Chaque clic de page recharge uniquement ce qui est nécessaire, ce qui :

- Allège le temps de chargement
- Facilite la lecture
- Permet de **naviguer de manière structurée**

La pagination est présente dans les listes d'œuvres et d'articles, et **elle respecte l'URL**, ce qui est bon pour le SEO.

Les tris et filtres : priorité à la clarté

Même sans base de données ultra complexe, j'ai prévu des options de tri :

- Les œuvres peuvent être séparées dès la vue en Films / Bandes dessinées
- Dans le back-office, les contenus sont triés par date, type ou auteur

Je prépare aussi l'ajout de filtres dynamiques supplémentaires (par genre, par année…) si j'ai le temps après la soutenance.

10. Technologies utilisées : simples, efficaces et durables

Un projet volontairement sans framework

J'ai choisi de **ne pas utiliser de framework** comme Laravel ou Symfony pour ce projet.
L'objectif était de **comprendre chaque ligne de mon code**, de l'écriture à la gestion des erreurs.
Cela m'a permis de :

- Créer mon propre routeur, adapté à mes besoins.
- Structurer manuellement le projet en MVC.
- Gérer les erreurs, les accès et les requêtes avec précision.

En formation, on m'a expliqué qu'apprendre à coder sans dépendre d'un framework était **une bonne base pour progresser ensuite**. Je confirme.

Front-End : clarté, réactivité, accessibilité

- **HTML5** : pour structurer proprement mes pages et optimiser l'accessibilité.
- **CSS3** : pour le style, bientôt transformé en **SASS (SCSS)** pour plus de modularité.
- **JavaScript Vanilla** : pas de bibliothèque externe, je gère toutes les interactions avec fetch, les modales, les carrousels, les sliders, etc.

Ce choix volontairement "léger" rend le site **rapide, lisible et compatible partout**.
J'ai appris à manipuler le DOM, à sécuriser les saisies utilisateur, à gérer les transitions et messages dynamiques sans recharger la page.

Back-End : stabilité, sécurité et contrôle

- **PHP 8** : langage principal du serveur, utilisé pour les contrôleurs, modèles, API internes…

- **MySQL** : gestion des données, relations (utilisateur, article, œuvre…), avec requêtes préparées.

- **PDO** : pour interagir proprement avec MySQL et sécuriser les échanges.

J'utilise aussi :

- **phpdotenv** : pour cacher mes identifiants de base de données dans un fichier .env

- **phpMyAdmin** : pour administrer les tables pendant le développement

- **XAMPP** : pour simuler un serveur Apache localement

11. Pourquoi ce choix de stack ?

Je viens d'un milieu manuel, je voulais quelque chose de solide

Avant d'être développeur, j'étais ouvrier. J'aime les choses **concrètes, stables et compréhensibles**.
J'ai choisi une stack simple, mais qui m'apprend les fondamentaux.

Chaque outil utilisé a une **raison précise** :

- HTML / CSS / JS : standard du web, aucun besoin de plugin ou d'environnement externe.

- PHP / SQL : parfait pour apprendre la logique serveur, avec des bases claires.

- PDO et les requêtes préparées : **essentiels pour la sécurité**, je les ai intégrés dès le début.

- MVC sans framework : pour **savoir vraiment ce que je fais**, et ne pas dépendre d'une magie extérieure.

Mon objectif était d'apprendre à **bâtir un projet de A à Z**, sans tricher. C'est ce que j'ai fait.

Un projet déployable et maintenable

J'ai pensé ce projet pour qu'il soit :

- Facile à déployer sur un hébergeur classique (o2switch)
- Facile à lire et à corriger (grâce aux commentaires et au découpage logique)
- Facile à faire évoluer (nouvelles fonctionnalités comme les favoris ou les notations)

En résumé, j'ai fait un choix de stack **pédagogique, mais réaliste**, qui peut grandir avec moi.

12. Architecture MVC : comment je l'ai comprise et utilisée

Une séparation claire entre les responsabilités

MVC signifie Modèle - Vue - Contrôleur. J'ai structuré mon site comme ça :

- **Les Modèles** : ils communiquent avec la base de données.
- **Les Contrôleurs** : ils reçoivent l'action demandée dans l'URL, appellent le bon modèle, préparent les données, et envoient tout à la vue.
- **Les Vues** : elles reçoivent uniquement les données nécessaires et affichent le HTML. Elles ne contiennent **aucune logique métier**, juste de l'affichage.

J'ai respecté le MVC à chaque étape

Quand un utilisateur visite une page, voilà ce qu'il se passe :

1. L'URL est interceptée par **mon routeur** (Router.php).
2. Je construis dynamiquement le **nom du contrôleur et de la méthode**.
3. Le contrôleur récupère les données via un **modèle**.
4. Ces données sont passées à une **vue** via View::render().

Je n'écris jamais de SQL dans les vues, ni de HTML dans les contrôleurs. Tout est séparé pour que ce soit **plus clair, plus facile à maintenir, et plus sécurisé**.

Une logique MVC que je peux expliquer, ligne par ligne

L'intérêt de coder sans framework, c'est que je **comprends chaque mécanisme** :

- Pourquoi j'utilise safeExecute() au lieu de query()
- Pourquoi je vérifie l'existence d'un ID avant d'afficher une fiche
- Pourquoi je gère les messages flash et les erreurs via ErrorHandler
- Pourquoi je ne redirige plus avec header() brut mais via une méthode du contrôleur

Je peux commenter chaque fichier, et je l'ai fait.
J'ai conçu ce site pour qu'il soit **compréhensible par moi dans 6 mois**, et **défendable devant un jury aujourd'hui**.

13. Sécurité : une priorité dès le début du projet

Des erreurs, des tests, et beaucoup d'apprentissages

Je savais que la sécurité allait être un point clé du projet, surtout en manipulant des données utilisateurs.
Je me suis donc **formé, documenté, testé, cassé et recommencé** plusieurs fois.

Ce que j'ai mis en place concrètement :

Protection contre les injections SQL

Toutes les requêtes en base utilisent des **requêtes préparées**.
J'ai centralisé ça dans BaseModel.php, via deux méthodes :

- safeExecute() pour insérer, mettre à jour ou supprimer
- safeQuery() pour faire des sélections simples

Ça me permet :

- de ne jamais concaténer une variable directement dans une requête SQL
- d'éviter les attaques de type '; DROP TABLE utilisateurs; --

Protection contre les failles XSS

- Tous les contenus affichés depuis la base sont passés par htmlspecialchars() pour éviter les scripts.
- Les commentaires en particulier sont protégés côté PHP et côté JavaScript.
- J'ai aussi échappé toutes les entrées dynamiques dans les scripts JS comme recherche.js.

Hashage des mots de passe

- À l'inscription, les mots de passe sont hashés avec password_hash().
- À la connexion, je vérifie le mot de passe avec password_verify().
- Je ne stocke jamais de mot de passe en clair.

Validation des formulaires

- Tous les champs sont vérifiés côté serveur : champs obligatoires, formats, types.
- J'ai aussi ajouté une vérification visuelle côté client pour améliorer l'expérience utilisateur (taille image, erreurs flash…).

Gestion des erreurs propre

- Aucune erreur technique n'est affichée à l'utilisateur.
- En cas de souci SQL, je déclenche une erreur 500 via ErrorHandler.
- Si une route n'existe pas ou si une œuvre n'est pas trouvée, j'affiche une page 404 claire.

14. Gestion des rôles et conformité RGPD

Des rôles bien définis dans tout le projet

Chaque utilisateur peut avoir un ou plusieurs rôles. Je gère ça dans la table appartenir.

- **Visiteur** : peut consulter les articles et œuvres, mais rien de plus.
- **Utilisateur connecté** : peut commenter, gérer son compte, etc.
- **Rédacteur** : peut ajouter/modifier ses propres articles ou œuvres, uniquement via le RedacteurPanel.
- **Administrateur** : a accès au panneau admin complet, peut modifier ou supprimer tout, et gérer les rôles des autres.

Vérification automatique dans les contrôleurs

- J'utilise AuthHelper::isAdmin() ou isRedacteur() dans chaque action sensible.
- Impossible d'accéder à une page protégée sans être logué ou autorisé.
- Les liens dans les headers s'adaptent au rôle (je ne montre pas ce que l'utilisateur ne peut pas faire).

Respect du RGPD

J'ai fait attention à ce que mon projet respecte les **principes essentiels du RGPD** :

- **Données minimales** : je ne demande que l'email et le mot de passe à l'inscription.
- **Consentement aux cookies** : via une bannière claire avec acceptation/refus, stockée en cookie.
- **Suppression des données** : un utilisateur peut désactiver son compte. Ses données sont alors invisibles, mais conservées pour éviter de casser les commentaires ou articles.
- **Aucune transmission de données** : je ne transmets rien à un service tiers

J'explique tout ça dans la **page de politique de confidentialité**, visible en bas de page.

15. Ce que j’ai appris, ce que j’ai traversé

Un projet technique… mais aussi personnel

Ce projet, ce n’était pas juste des lignes de code.
C’était pour moi **le test final** de ma reconversion.
Je suis passé de **manutentionnaire** à développeur web. Et pour ça, il m’a fallu plus que de la logique.

J’ai appris à :

- **Structurer un projet complet** : base de données, MVC, routes, sécurité, vues, etc.
- **Me débrouiller seul** : chercher, comprendre, corriger.
- **Commenter clairement mon code** : pour moi et pour le jury.
- **Prendre des décisions de développement** : quoi factoriser, quoi simplifier, quoi tester.
- **Rester cohérent** : ne pas réinventer la roue à chaque fois, mais faire simple et propre.
- **Documenter et sécuriser chaque fichier** : je me suis mis dans la peau d’un collègue qui lirait mon code plus tard.

J’ai aussi traversé :

- Des moments de découragement (bugs incompréhensibles)
- Des erreurs de logique (routes mal interprétées, variables vides…)
- Des questions sans réponse immédiate (“Pourquoi ma modale ne s’ouvre plus ?”)
- Des nuits à revoir mes commentaires, à simplifier mes explications

Mais j’ai tenu bon. Et aujourd’hui, je peux dire que **ce site, je le comprends, je l’assume, et je peux le défendre**.

16. Accessibilité, performances et responsive design

Tests d'accessibilité

J'ai utilisé plusieurs outils pour m'assurer que le site était **lisible, navigable et compréhensible** par tous :

- **Google page insight** : pour identifier les problèmes d'accessibilité visuelle et structurelle.

- **Navigation clavier seule** : tous les boutons, menus, formulaires peuvent être utilisés sans souris.

- **Contrastes vérifiés** : les couleurs choisies (issues du moodboard) sont testées pour garantir la lisibilité.

Je me suis appuyé sur les recommandations WCAG, avec une attention portée aux titres hiérarchisés (h1, h2, etc.), aux balises alt, aux labels de formulaire, et aux éléments ARIA dans les modales et menus.

Responsive design

Le site est **conçu mobile-first** :
→ d'abord pensé pour les petits écrans, avec une mise en page fluide et réorganisée sur tablette et desktop.

Les menus s'adaptent, les blocs se superposent intelligemment, et les sliders deviennent scrollables.
Aucune fonctionnalité n'est cachée sur mobile.

→ Le travail en SCSS est en cours pour optimiser encore cette partie et faciliter la gestion des media queries.

Performances

J'ai testé le site avec :

- **Google PageSpeed Insights**

- **EcoIndex.fr**

Les résultats sont globalement bons.
→ J’ai réduit le poids des images (WebP, compression)
→ J’ai utilisé le lazy loading pour ne charger que ce qui est nécessaire à l’écran
→ Je prévois de minifier les scripts CSS/JS
→ Et de limiter les requêtes HTTP externes

Tout cela contribue à un **site rapide et léger**, même en mobile ou sur une connexion lente.

17. SEO et indexation

Structure HTML optimisée

Chaque page est structurée pour être **facilement indexée par les moteurs de recherche** :

- Balises title et meta description spécifiques à chaque vue (A PREVOIR, seule la page d'accueil est faite pour le moment)
- Balises alt sur toutes les images
- URLs propres grâce au .htaccess et au routeur MVC
- Hierarchie claire avec balises Hn

Fichiers d'aide au référencement

Deux fichiers essentiels SERONT été créés à la racine du projet (manque de temps) :

- **robots.txt** : indique aux robots des moteurs ce qu'ils peuvent explorer ou non.
- **sitemap.xml** : généré automatiquement pour lister toutes les pages accessibles du site.

Google Search Console / Analytics

Je prévois de connecter le site à Google Search Console pour :

- suivre l'indexation réelle du site
- détecter les erreurs d'exploration
- suivre les requêtes de recherche liées au site

Google Analytics permettra de **mesurer les comportements utilisateurs** : pages les plus consultées, taux de rebond, etc.
→ Cela me servira plus tard à **adapter le contenu ou l'arborescence** en fonction des usages.

18. Évolutions futures possibles

Le site est pensé pour évoluer. J'ai volontairement codé une base solide et extensible.

Voici les **fonctionnalités qui pourraient être ajoutées** dans une future version :

Suppression de compte via demande

Actuellement, l'utilisateur peut désactiver son compte dans son profil.
Mais à terme, une **demande de suppression définitive** pourra être transmise à un administrateur via un formulaire sécurisé.

Barre de recherche dans les panels Admin / Rédacteur

Pour gagner du temps, une barre de recherche AJAX pourrait permettre :

- de retrouver rapidement une œuvre ou un article
- de filtrer les utilisateurs, les commentaires, etc. Cela rendrait la gestion de contenu encore plus fluide.

Système de notation

Chaque utilisateur pourrait **noter une œuvre** sur 5 étoiles.
→ Les moyennes s'afficheraient sur les fiches œuvres, avec des avis facultatifs.

Cela renforcerait l'aspect communautaire, en ajoutant une dimension qualitative au site.

Favoris

Un utilisateur connecté pourrait ajouter une œuvre ou un article à ses **favoris**.
→ Ces favoris seraient visibles dans son profil, et stockés en base.

Forum intégré

Un petit **forum communautaire** pourrait être intégré :

- pour discuter des œuvres
- proposer des analyses
- ou suggérer de nouveaux contenus

Cela créerait un espace vivant autour du projet Ciné-Hurlant.

19. Conclusion : un projet complet, un parcours transformé

Ce projet Ciné-Hurlant représente beaucoup plus qu'un simple exercice de développement.

Il incarne **ma reconversion**, **mon envie d'apprendre**, **de comprendre**, et de créer un site **cohérent, solide, lisible et utile**.

J'ai appris à travailler avec :

- une architecture MVC que je comprends de bout en bout
- une base de données relationnelle complète
- du PHP moderne avec gestion des erreurs et sécurité
- JavaScript dynamique avec fetch() et modales personnalisées
- une logique utilisateur pensée pour l'ergonomie et la sécurité

J'ai aussi découvert le plaisir de **documenter, structurer, refactorer, commenter** tout mon travail.
→ Non pas pour les autres, mais pour être sûr **moi-même de tout comprendre**.

J'ai beaucoup progressé techniquement.
Mais j'ai aussi gagné en **confiance**.
Aujourd'hui, je sais **ce que je fais, pourquoi je le fais, et comment le faire évoluer**.

Ciné-Hurlant, c'est mon premier projet de développeur.
Et je suis fier de pouvoir le présenter au jury, **non pas comme un site "fait", mais comme une base vivante, stable, et maîtrisée.**