P\_WEB295  
Passion Lecture



Premat Luca

Skupovska Veronika

Almeida Nelson

ETML, Lausanne

Vennes

24 périodes

M. G. Charmier

Table des matières

[Introduction 3](#_Toc211211602)

[1. Planification du projet 4](#_Toc211211603)

[2. Analyse de l’API REST 4](#_Toc211211604)

[3. Analyse de la base de données 5](#_Toc211211605)

[4. Analyse de la structure du code 6](#_Toc211211606)

[5. Schéma de l’architecture 7](#_Toc211211607)

[6.1. Authentification et gestion des rôles 8](#_Toc211211608)

[6.2. Sécurité 8](#_Toc211211609)

[6.3. Fonctionnalités techniques réalisées 9](#_Toc211211610)

[7. Test 10](#_Toc211211611)

[8. Conclusion 11](#_Toc211211612)

[8.1. Organisation du code et gestion de version 11](#_Toc211211613)

[8.2. Conclusion générale 11](#_Toc211211614)

[8.3. Conclusion personnelle et critique constructive 11](#_Toc211211615)

[9. Webographie / Glossaire / Annexe 12](#_Toc211211616)

[Webographie 12](#_Toc211211617)

[Glossaire 12](#_Toc211211618)

[Annexe 12](#_Toc211211619)

# Introduction

Le projet *Passion Lecture* s’inscrit dans le cadre du module P\_WEB295. Il consiste à développer le backend d’une application web permettant aux utilisateurs de partager et consulter des ouvrages littéraires. L’objectif est de mettre en place une API REST complète qui gère les principales fonctionnalités liées aux livres, aux auteurs, aux catégories et aux utilisateurs, tout en respectant les bonnes pratiques de développement vues en cours.

Cette API doit permettre à un futur frontend de consommer ses données de manière simple, sécurisée et structurée. Ainsi, le backend assurera la création, la modification et la suppression d’ouvrages, la gestion des appréciations et des commentaires, ainsi que l’authentification des utilisateurs et la distinction des rôles (utilisateur, administrateur).

Le projet est également l’occasion de consolider des compétences techniques spécifiques :

* La conception et l’organisation d’une base de données relationnelle adaptée.
* L’utilisation du framework AdonisJS avec l’ORM Lucid pour gérer la persistance des données.
* L’implémentation de bonnes pratiques de sécurité (authentification par jetons, gestion des droits d’accès, règles CORS).
* La validation des données et la gestion cohérente des statuts HTTP.
* La documentation et le test des routes à l’aide d’outils tels que Swagger et Bruno.

Un accent est mis sur l’écoconception Web, afin de réduire l’impact environnemental de l’application, et sur l’organisation du travail collaboratif à l’aide d’outils comme GitHub (gestion de version, suivi des tâches via Projets).

Ce rapport présente l’analyse, la réalisation et les tests effectués, puis conclut sur les aspects organisationnels et les perspectives d’amélioration du projet.

# 1. Planification du projet

La planification du projet a été effectuée à l’aide de GitHub Projects, selon une organisation Kanban comportant trois colonnes : *À faire*, *En cours* et *Terminé*. Ce système a permis de visualiser facilement l’avancement des différentes tâches et de répartir le travail de manière équilibrée.

Chaque membre du groupe — Veronika, Luca et Nelson — s’est vu attribuer la responsabilité complète d’une ou plusieurs tables, comprenant la création du modèle, de la migration, du validateur, du contrôleur, des routes et du seeder associés.  
Les tâches couvraient ainsi l’ensemble des entités principales de l’application, garantissant une progression parallèle et coordonnée du développement.

Cette approche a favorisé une bonne visibilité sur le suivi du projet et une collaboration efficace tout au long du développement.

# 2. Analyse de l’API REST

L’API REST du projet *Passion Lecture* a été conçue pour permettre la gestion complète des ressources principales de l’application : les livres, les auteurs, les catégories et les utilisateurs. Elle repose sur le principe du CRUD (Create, Read, Update, Delete), garantissant une structure claire et homogène pour toutes les entités.

Chaque ressource dispose de ses propres routes REST définies dans le fichier de configuration routes.ts. Ces routes utilisent les verbes HTTP standards — GET, POST, PUT et DELETE — permettant respectivement la consultation, la création, la modification et la suppression de données. Ce découpage assure une séparation logique et cohérente entre les différentes opérations.

Les échanges entre le client et le serveur s’effectuent en format JSON, ce qui facilite l’intégration avec un futur frontend et garantit une communication simple et lisible. Les réponses de l’API respectent les conventions HTTP, notamment les statuts 200 pour les lectures réussies, 201 pour les créations, 204 pour les suppressions, ainsi que 404 et 422 pour les erreurs de recherche ou de validation.

L’API repose sur l’ORM Lucid d’AdonisJS, qui gère la persistance des données et les relations entre les tables. Les liens logiques entre les entités ont été établis à l’aide des décorateurs belongsTo et hasMany, par exemple pour associer un livre à un auteur, une catégorie ou un utilisateur, ou encore pour relier les commentaires et évaluations aux livres correspondants. Cette approche permet d’effectuer des requêtes complexes tout en conservant une structure claire, lisible et conforme aux bonnes pratiques du développement REST.

Dans son ensemble, cette API assure une interaction fluide entre le serveur et un futur client web, tout en garantissant la cohérence, la sécurité et la maintenabilité du backend.

# 3. Analyse de la base de données

La base de données du projet *Passion Lecture* a été conçue selon un modèle relationnel visant à garantir la cohérence et la fiabilité des informations échangées entre les différentes entités. La conception suit une démarche progressive allant du modèle conceptuel (MCD) au modèle logique (MLD), puis au modèle physique (MPD) implémenté dans MySQL via les migrations AdonisJS.

Le modèle conceptuel des données (MCD) comporte six entités principales : *Book*, *Writer*, *Category*, *User*, *Comment* et *Evaluate*.  
Les relations définies traduisent les interactions fonctionnelles du système.  
Chaque livre appartient à une catégorie et est écrit par un auteur unique.  
Un utilisateur peut publier plusieurs ouvrages et laisser plusieurs commentaires, mais ne peut donner qu’une seule évaluation par livre. Ces contraintes métier sont assurées par les associations *comment* et *evaluate*, qui relient les entités *Book* et *User*.  
Ainsi, la relation entre *User* et *Book* est de type 1–N, tandis que les relations *Comment* et *Evaluate* introduisent des dépendances supplémentaires permettant de gérer les interactions sociales (avis et notes).

Le modèle logique (MLD) traduit ces relations en tables concrètes adaptées à MySQL.  
La table des livres (*t\_book*) contient des clés étrangères vers les tables *t\_category*, *t\_writer* et *t\_user*, exprimant respectivement la catégorie, l’auteur et le créateur du livre.  
Les tables *t\_comment* et *t\_evaluate* gèrent les relations entre utilisateurs et livres :

* *t\_comment* relie un utilisateur à un livre avec un texte de commentaire,
* *t\_evaluate* relie un utilisateur à un livre avec une note numérique.

Enfin, le modèle physique (MPD) est généré à travers les migrations AdonisJS, assurant la création automatique des clés primaires et étrangères, ainsi que la mise en place des contraintes d’intégrité référentielle.  
Cette structure relationnelle claire et normalisée garantit la cohérence des données, facilite l’évolution du projet et soutient efficacement la logique de l’API REST mise en place.

# 4. Analyse de la structure du code

L’architecture du code du projet *Passion Lecture* suit la logique imposée par le framework AdonisJS, qui repose sur une organisation claire, modulaire et orientée MVC (*Model-View-Controller*).  
Même si aucune interface visuelle n’a été développée dans ce projet, cette structure reste essentielle pour séparer la logique métier, la gestion des données et la définition des routes.

La couche modèle (Models) regroupe les définitions des entités principales : *Book*, *Writer*, *Category*, *User*, *Comment* et *Evaluate*.  
Chaque modèle correspond directement à une table de la base de données et utilise l’ORM Lucid pour assurer la correspondance entre les objets JavaScript et les enregistrements MySQL.  
Les décorateurs @hasMany et @belongsTo permettent de décrire précisément les relations entre les entités. Par exemple, un utilisateur peut posséder plusieurs livres et plusieurs commentaires, tandis qu’un livre appartient à un auteur, une catégorie et un utilisateur spécifique. Cette approche garantit la cohérence des interactions entre les différentes ressources de l’API.

La couche contrôleur (Controllers) gère la logique applicative et les opérations liées aux requêtes HTTP.  
Chaque entité dispose de son propre contrôleur dédié, chargé d’implémenter les fonctionnalités CRUD : affichage, création, modification et suppression.  
Ces contrôleurs interagissent directement avec les modèles et renvoient des réponses structurées au format JSON. L’usage systématique des méthodes asynchrones (async/await) assure une gestion fluide et sécurisée des appels à la base de données.

La couche validation (Validators) est utilisée pour vérifier la conformité des données reçues avant leur insertion ou leur modification.  
Grâce au module VineJS, chaque entité dispose d’un validateur définissant les règles de validation (longueur minimale des chaînes, format des champs, types numériques, etc.).  
Cela permet d’éviter l’injection de données invalides et renforce la fiabilité du backend.

Enfin, la couche base de données (Migrations, Seeders et Factories) facilite la gestion et la réinitialisation du schéma.  
Les *migrations* créent automatiquement les tables et leurs relations, les *seeders* injectent des données de test cohérentes, et les *factories* génèrent des entrées fictives à l’aide de la bibliothèque *Faker*.  
Cette structure assure la reproductibilité du projet et simplifie les tests.

Dans son ensemble, le code suit une organisation claire et évolutive, conforme aux bonnes pratiques de développement backend. Cette structure modulaire rend le projet facilement maintenable, tout en garantissant la cohérence entre les différentes composantes de l’API.

# 5. Schéma de l’architecture

L’architecture du projet *Passion Lecture* s’appuie sur une organisation en couches typique des applications web modernes, séparant clairement la logique serveur, la gestion des données et l’interface utilisateur.  
Dans cette première phase, le développement s’est concentré sur la partie backend, tandis que le frontend sera réalisé ultérieurement pour consommer l’API REST déjà opérationnelle.

Le backend, développé avec AdonisJS, constitue le cœur du système.  
Il gère la logique applicative, la validation des données et les échanges avec la base MySQL.  
Les routes définissent les points d’accès de l’API et orientent chaque requête vers le contrôleur approprié.  
Ces contrôleurs traitent les opérations CRUD (création, lecture, mise à jour, suppression), appliquent les règles de validation et renvoient les réponses au format JSON, facilitant ainsi la communication avec un futur client web.

L’ORM Lucid joue un rôle essentiel dans cette architecture.  
Il permet de manipuler les entités du projet (*Book*, *Writer*, *Category*, *User*, *Comment* et *Evaluate*) sous forme d’objets tout en automatisant les interactions avec la base de données relationnelle.  
Les relations 1-N et N-N sont gérées directement dans les modèles grâce aux décorateurs @hasMany et @belongsTo, assurant une structure claire et cohérente.

La base de données MySQL assure la persistance des informations.  
Elle a été mise en place à l’aide des *migrations*, *seeders* et *factories* d’AdonisJS, garantissant une création rapide et fiable des tables et de leurs relations.  
Cette organisation permet de réinitialiser et de tester facilement le projet pendant la phase de développement.

L’ensemble de cette architecture a été conçu pour être évolutif :  
le backend peut dès à présent être consommé par un futur frontend (web ou mobile), et des outils comme Swagger ou Postman pourront être intégrés ultérieurement pour la documentation et les tests.  
Cette séparation claire des responsabilités assure une base solide, maintenable et adaptée à la poursuite du projet.

## 6.1. Authentification et gestion des rôles

## 6.2. Sécurité

La sécurité a été abordée à trois niveaux : validation des données, statuts/erreurs HTTP, et politique CORS.

* Validation des données. Les entrées utilisateur sont décrites par des validateurs VineJS (ex. userValidator) afin d’encadrer types, longueurs et formats avant insertion/mise à jour. Cette approche évite l’injection de données invalides et correspond aux bonnes pratiques vues au cours. À intégrer systématiquement via request.validateUsing(...) dans les points d’entrée critiques (création/mise à jour d’utilisateurs).
* Statuts & gestion d’erreurs. Les contrôleurs utilisent des réponses explicites cohérentes avec REST (200/OK pour lecture, 201/Created pour création, 204/No Content pour suppression). Les recherches par id s’appuient sur findOrFail, ce qui déclenche automatiquement une 404 Not Found si la ressource n’existe pas ; les erreurs de validation doivent remonter en 422 Unprocessable Entity via les validateurs.
* CORS. La politique CORS est activée avec origine autorisée, méthodes standards (GET/HEAD/POST/PUT/DELETE) et support des *credentials*. Cela prépare l’API à être consommée par un futur frontend tout en contrôlant les échanges inter-origines.
* Mots de passe & rôles (préparés). Le modèle User intègre déjà le hachage (Scrypt) et withAuthFinder, et masque le champ de mot de passe en sortie ; l’authentification par jetons et l’usage de isAdmin seront activés ultérieurement pour protéger les routes sensibles. (Voir 6.1.)

Axes d’amélioration immédiats (court terme).  
Appliquer systématiquement la validation dans les méthodes store/update des ressources sensibles (notamment *User*), ajouter une politique de messages d’erreurs homogènes, et préparer le verrouillage de certaines routes en lecture/écriture dès l’activation de l’authentification.

## 6.3. Fonctionnalités techniques réalisées

La première phase du projet *Passion Lecture* a permis de mettre en place l’ensemble des fonctionnalités techniques liées au backend.  
Les routes REST ont été définies pour toutes les entités principales (*Book*, *Writer*, *Category*, *User*), chacune disposant d’un CRUD complet (création, lecture, modification, suppression).  
Les contrôleurs assurent la logique métier et utilisent les méthodes de l’ORM Lucid pour manipuler les données.  
Les réponses HTTP respectent les conventions REST, et les données sont échangées au format JSON.

Les relations entre tables ont également été intégrées afin d’assurer la cohérence du modèle :  
un livre appartient à un auteur, à une catégorie et à un utilisateur ;  
un utilisateur peut posséder plusieurs livres et plusieurs commentaires.  
Ces relations sont définies à l’aide des décorateurs @hasMany et @belongsTo, permettant de charger les données associées grâce au préchargement Lucid.

Les migrations, seeders et factories ont été utilisés pour automatiser la création de la base et générer des données de test.  
Ce mécanisme a facilité les vérifications dans Bruno et le bon fonctionnement des routes CRUD.

Cette première étape a permis d’obtenir une API REST stable, complète et conforme aux objectifs du module, servant de base solide pour la suite du projet, notamment l’intégration de l’authentification et du futur frontend.

# 7. Test

Les tests du backend ont été effectués à l’aide du logiciel Bruno, un outil dédié à la vérification des API REST.  
Il a permis de simuler des requêtes HTTP (GET, POST, PUT et DELETE) et de contrôler la validité des réponses renvoyées par le serveur.

Chaque entité principale (*Book*, *Writer*, *Category* et *User*) a été testée à travers son ensemble de routes CRUD.  
Les scénarios de test consistaient à créer de nouvelles entrées, à les modifier, à les consulter puis à les supprimer, afin de vérifier la cohérence des opérations et des statuts HTTP retournés (200, 201, 204, 404, 422).

Ces essais ont également permis de confirmer le bon fonctionnement de la base de données et des relations entre les tables.  
Les données validées ou invalides ont été testées afin de s’assurer que les règles de validation étaient correctement appliquées et que les erreurs étaient gérées de manière cohérente.

Dans l’ensemble, les résultats obtenus ont démontré la stabilité et la fiabilité du backend, garantissant le bon comportement de l’API en conditions normales d’utilisation.

# 8. Conclusion

## 8.1. Organisation du code et gestion de version

Le travail collaboratif a été organisé à l’aide de GitHub, utilisé à la fois pour le versionnement du code et pour le suivi de l’avancement à travers GitHub Projects.  
Chaque membre du groupe (Nelson, Veronika et Luca) a travaillé sur des branches dédiées, avant fusion dans la branche principale après validation du code.  
Cette méthode a permis d’assurer une bonne traçabilité des modifications et de réduire les conflits lors des intégrations.  
Les commits étaient nommés de manière claire et régulière, reflétant les étapes du développement (création des modèles, contrôleurs, seeders, validations, etc.).

## 8.2. Conclusion générale

Cette première phase du projet *Passion Lecture* a permis de concevoir et d’implémenter un backend complet, fonctionnel et conforme aux objectifs du module.  
Toutes les routes principales ont été développées, les relations entre entités sont opérationnelles et les tests effectués ont validé la stabilité de l’API.  
Le projet offre ainsi une base solide pour la suite du développement, notamment l’intégration de l’authentification et du futur frontend.  
Ce travail a permis de mettre en pratique des compétences clés telles que la structuration d’une API REST, la modélisation relationnelle, la validation des données et l’utilisation d’un ORM moderne comme Lucid.

## 8.3. Conclusion personnelle et critique constructive

Ce projet a représenté une étape importante dans l’apprentissage du développement web backend.  
Il a permis de mieux comprendre la logique d’AdonisJS et le rôle de chaque couche du framework, tout en renforçant les compétences en conception de base de données et en organisation du code.  
Le travail en groupe a aussi mis en évidence l’importance d’une communication claire et d’une planification rigoureuse.  
Pour la suite, il serait souhaitable d’améliorer la coordination dans la répartition des tâches et de planifier plus de temps pour les tests approfondis et la documentation technique.  
L’ajout de l’authentification, la mise en place du frontend et la documentation Swagger constitueront les prochaines étapes naturelles du projet.

# 9. Webographie / Glossaire / Annexe

## Webographie

* Documentation officielle AdonisJS : <https://docs.adonisjs.com>
* Documentation VineJS (validation) : https://vinejs.dev
* Documentation MySQL : https://dev.mysql.com/doc
* Guide HTTP status codes : https://developer.mozilla.org/fr/docs/Web/HTTP/Status
* Outil Bruno – REST Client : https://www.usebruno.com

## Glossaire

* API REST : Interface permettant la communication entre un client et un serveur via les méthodes HTTP standard (GET, POST, PUT, DELETE).
* CRUD : Acronyme de Create, Read, Update, Delete — opérations de base sur les données.
* ORM (Object Relational Mapping) : Outil permettant de manipuler des données relationnelles via des objets.
* Migration / Seeder / Factory : Mécanismes d’AdonisJS permettant respectivement de créer les tables, d’insérer des données initiales et de générer des données fictives.
* CORS (Cross-Origin Resource Sharing) : Mécanisme de sécurité contrôlant les requêtes HTTP provenant d’origines externes.
* Validation : Processus consistant à vérifier que les données saisies respectent des règles définies avant leur enregistrement.

Annexe

Utilisation de l’intelligence artificielle :

Une assistance ponctuelle par ChatGPT (OpenAI) a été utilisée pour la rédaction du rapport, la reformulation de certaines sections techniques et la vérification de la cohérence du contenu.  
Les codes et structures présentés dans le projet ont cependant été réalisés et compris par les membres du groupe.  
L’IA n’a pas remplacé la compréhension ni l’exécution du travail, mais a servi d’outil d’aide à la rédaction et à la mise en forme.