module de connection cda

1. Présentation du projet / Genèse

1.1 Besoin initial exprimé

Lorsque j'ai entamé ce projet, mon objectif principal était de concevoir une **API sécurisée** capable de gérer les opérations classiques d'authentification et de gestion utilisateur, tout en posant les fondations d'un backend réutilisable pour des projets plus complexes. L'idée de départ m'est venue à la fois d'un besoin pédagogique — dans le cadre de ma formation CDA, où je devais démontrer ma capacité à sécuriser une API — et d'une volonté personnelle de maîtriser les problématiques de sécurité web dès la conception d'un backend.

Plutôt que de créer un projet artificiel, j'ai préféré m'inspirer d'un **besoin courant en entreprise** : disposer d'une **API fiable**, **maintenable** et **sécurisée**, sur laquelle une application web ou mobile pourrait venir se connecter. Cela impliquait la gestion d'utilisateurs, l'authentification, la mise à jour sécurisée des données sensibles comme l'email ou le mot de passe, et la possibilité de supprimer son compte. Il était essentiel pour moi d'intégrer dès le départ des mécanismes comme le **chiffrement des mots de passe avec bcrypt**, la **protection des routes via JWT**, et une **vérification d'identité systématique** pour toute opération critique.

1.2 Objectifs généraux

Dans ce cadre, je me suis fixé plusieurs objectifs concrets et ambitieux. Il ne s'agissait pas simplement de créer une API fonctionnelle, mais une **API robuste**, conçue selon les bonnes pratiques du développement backend.

Je voulais permettre à un utilisateur de s'inscrire et de se connecter de manière sécurisée, avec **un système de rôle** (utilisateur simple ou

administrateur) permettant de **restreindre l'accès à certaines routes**. J'ai également choisi d'**organiser mes routes en suivant les conventions REST**, de façon à ce que chaque ressource (utilisateur, jeux, etc.) dispose de ses propres endpoints, bien séparés et logiquement structurés.

Un autre objectif important était d'**intégrer des tests** — à la fois fonctionnels (via Postman) et unitaires (avec Jest) — pour fiabiliser le comportement de l'API. Enfin, comme j'envisageais de réutiliser cette API dans un projet React côté frontend, il était impératif que la **logique métier soit découplée de toute logique de présentation**.

1.3 Cahier des charges

Pour cadrer ce projet, je me suis imposé un **cahier des charges clair et structurant**. J'ai choisi d'implémenter une **architecture RESTful**, bâtie sur **Node.js avec Express**, et une **base de données relationnelle MySQL**. Le choix de JWT s'est imposé naturellement pour l'authentification, en raison de sa simplicité d'intégration et de sa compatibilité avec des systèmes de frontend déconnectés comme React.

Chaque mot de passe devait être **hashé avec bcrypt** avant stockage, afin de respecter les standards de sécurité. J'ai également mis en place un système de **middlewares** pour centraliser la protection des routes, et un gestionnaire d'erreurs global afin de fournir des réponses cohérentes à l'utilisateur ou au développeur.

Le projet devait également inclure une **documentation** (via Postman et un fichier README), des **tests unitaires ciblés**, et une **structure de fichiers claire**, organisée en `controllers`, `routes`, `middlewares`, `models` et `utils`, de façon à faciliter la maintenance et la scalabilité future.

2. Environnement du projet

2.1 Architecture technique

Pour la conception de cette API, j'ai opté pour une architecture classique mais éprouvée, centrée sur le framework **Express.js**. Ce choix s'est imposé naturellement compte tenu de sa légèreté, de sa flexibilité et de sa large adoption dans le développement de services backend en JavaScript. Express m'a permis de structurer rapidement mes routes et de gérer facilement les middlewares, indispensables pour l'authentification et la gestion des erreurs.

Côté persistance des données, j'ai choisi **MySQL** comme système de base relationnelle. C'est un moteur que je maîtrise déjà, robuste et parfaitement adapté pour stocker des entités comme les utilisateurs ou les jeux, avec des relations bien définies. Pour l'authentification, j'ai décidé d'implémenter des **tokens JWT (JSON Web Tokens)**, car ils offrent une méthode stateless, sécurisée et bien adaptée à une API REST. Enfin, pour assurer la fiabilité de mon application, j'ai mis en place des tests via **Postman** (pour les tests manuels et de collection) ainsi que **Jest** pour des tests unitaires ciblés sur certaines fonctions.

Cette architecture m'a permis de maintenir une séparation claire entre les différentes couches (routes, contrôleurs, modèles, middlewares) et de garantir une base propre, évolutive et sécurisée.

2.2 Technologies et outils utilisés

Le projet repose sur **Node.js**, qui est à la base du runtime JavaScript utilisé pour faire tourner l'application côté serveur. Sa rapidité et sa nature non bloquante sont de vrais atouts pour un service d'API.

Le cœur de l'application repose sur **Express**, pour la gestion des routes et des requêtes HTTP. Pour sécuriser les mots de passe, j'ai utilisé **bcrypt**, une bibliothèque de hachage très réputée dans le domaine de la cybersécurité. Le

hachage est une étape cruciale car il permet de ne jamais stocker les mots de passe en clair dans la base de données, ce qui serait une faille critique.

L'authentification repose sur **jsonwebtoken (JWT)**, qui m'a permis d'implémenter un système de session sécurisé sans avoir à stocker quoi que ce soit côté serveur. Chaque utilisateur reçoit un token qu'il utilise pour prouver son identité à chaque requête sensible. Pour la configuration des variables d'environnement, notamment les clés secrètes, j'ai utilisé **dotenv**, ce qui m'a permis de garder les données sensibles hors du code source.

Pour tester les endpoints, j'ai travaillé avec **Postman**, qui m'a permis de vérifier chaque scénario de requête HTTP. J'ai également commencé à intégrer des tests unitaires avec **Jest**, dans le but d'automatiser certaines vérifications à terme. Ces outils m'ont permis de travailler plus efficacement, de détecter rapidement les erreurs, et de fiabiliser mon code.

3. Conception

3.1 Prototypage

Avant même d’écrire une ligne de code, j’ai tenu à poser les bases de mon API sur un plan clair. Cela a commencé par le **schéma de la base de données**, que j’ai conçu autour d’une structure relationnelle simple, mais extensible. J’ai créé une table `users`, contenant les champs essentiels comme `id`, `username`, `email`, `password` (haché bien sûr), et un champ `role` pour distinguer les administrateurs des utilisateurs standards. J’ai également anticipé d’éventuelles évolutions en prévoyant des clés étrangères dans d'autres tables, comme une table `games` pour des mini-jeux ou des scores liés aux utilisateurs.

Côté routes, j’ai dessiné une **architecture RESTful** : chaque ressource (user, auth, jeux, etc.) dispose de ses propres endpoints (`/api/users`, `/api/auth`, `/api/games`, etc.), et chaque action (GET, POST, PUT, DELETE) correspond à un comportement précis et cohérent. Cette organisation m’a permis de garder une logique claire et prévisible, tout en respectant les bonnes pratiques du développement d’API.

Ce travail préparatoire m’a énormément aidé à structurer le projet, à éviter les erreurs de modélisation et à anticiper les vérifications de sécurité nécessaires à chaque point d’entrée de l’API.

3.2 Identité visuelle

Ce projet étant centré sur le **backend**, je n’ai pas développé d’identité visuelle poussée à ce stade. Mon objectif était avant tout de produire un backend robuste, sécurisé et bien documenté, qui puisse éventuellement être consommé par une application front-end dans un second temps. Toutefois, j’ai

tenu à garder un minimum de lisibilité et de clarté dans les réponses JSON de l'API, en structurant bien les objets retournés et en utilisant des messages explicites pour les erreurs.

Dans une éventuelle version future, je prévois de créer une interface utilisateur avec **React**, qui viendrait consommer cette API et proposer une expérience graphique plus aboutie.

3.3 Structure de navigation

L'une de mes priorités a été de rendre l'API facilement **navigable et compréhensible**, notamment pour d'autres développeurs qui pourraient vouloir l'utiliser. Pour cela, j'ai structuré la navigation autour de **routes REST classiques**.

Par exemple, pour gérer les utilisateurs, j'ai prévu :

- `POST /api/auth/register` pour l'inscription,
- `POST /api/auth/login` pour la connexion,
- `GET /api/users/me` pour accéder à ses propres informations,
- `PUT /api/users/update-email` ou `/update-password` pour modifier ses données,
- `DELETE /api/users/delete-account` pour supprimer son compte.

Chaque route passe par des middlewares spécifiques : un middleware d'authentification pour vérifier le token JWT, et parfois un middleware de rôle pour restreindre certaines actions aux administrateurs. Grâce à cette

structure, j’ai pu centraliser la logique métier tout en maintenant la lisibilité et la maintenabilité du projet.

4. Réalisation

4.1 Version initiale

Lorsque j'ai commencé à développer l'API, j'ai choisi de procéder par étapes, en construisant d'abord une **version fonctionnelle minimale**, centrée sur l'authentification. J'ai mis en place les fonctionnalités essentielles : l'inscription (`/register`), la connexion (`/login`) et la protection des routes sensibles à l'aide d'un **token JWT**.

L'objectif à ce stade était de m'assurer que chaque nouvelle session utilisateur était bien gérée de manière sécurisée : j'ai utilisé la bibliothèque `jsonwebtoken` pour générer les tokens et `bcrypt` pour hacher les mots de passe. Ces outils, que j'ai sélectionnés pour leur maturité et leur large adoption, m'ont permis de mettre en place une authentification robuste sans réinventer la roue.

J'ai également structuré le code en suivant une architecture MVC simplifiée : les **controllers** contiennent la logique métier, les **routes** définissent les chemins d'accès, et les **middlewares** gèrent l'authentification ou les erreurs. Cette séparation des responsabilités m'a permis d'avoir un projet clair et maintenable dès le début.

4.2 Améliorations successives

Une fois les fondations posées, j'ai rapidement identifié plusieurs axes d'amélioration. J'ai d'abord travaillé sur la **gestion des erreurs** pour que l'API retourne toujours des messages clairs, cohérents, et dans un format structuré (JSON). Cela m'a permis de faciliter les tests, mais aussi d'anticiper une intégration front-end future.

Ensuite, j'ai renforcé la **logique métier**, en ajoutant des fonctionnalités comme la mise à jour d'un email ou d'un mot de passe, et la suppression d'un compte utilisateur. Pour ces actions sensibles, j'ai systématiquement exigé

une authentification via JWT, et dans certains cas, une confirmation (par exemple, saisie du mot de passe actuel avant de modifier un mot de passe).

J'ai aussi progressivement **nettoyé et refactoré le code** : suppression des doublons, réutilisation de fonctions utilitaires, centralisation de la logique d'authentification, etc. Cela m'a appris à faire évoluer mon code sans le casser, tout en le rendant plus clair.

Enfin, j'ai ajouté un système rudimentaire de **rôles** pour distinguer les utilisateurs simples des administrateurs. Cela ouvre la voie à des fonctionnalités plus avancées (modération, statistiques, etc.) dans des versions ultérieures.

4.3 Mise en production

Le projet étant pour l'instant en phase de développement, je l'ai **mis en production localement** sur mon environnement WAMP et via Node.js avec des ports dédiés pour le frontend et le backend. J'ai pris soin de bien séparer les fichiers `.env` contenant les informations sensibles (clé JWT, configuration MySQL) pour éviter toute fuite lors d'un éventuel déploiement.

J'ai également configuré les **en-têtes CORS**, afin de permettre les échanges entre le frontend (React, en cours de développement) et l'API backend, tout en limitant les origines autorisées. Ce point m'a semblé important dès le départ, car les mauvaises configurations CORS sont une faille fréquente dans les projets full-stack.

Je prévois, dans une prochaine étape, un **déploiement distant**, sans doute via un serveur VPS ou une plateforme comme Render ou Railway, avec base de données distante, pour simuler un environnement de production réel.

5. Collaboration et accompagnement

5.1 Collaboration

Ce projet a été conçu en **autonomie**, dans une logique d'apprentissage personnel et de mise en pratique de compétences acquises durant ma formation de Concepteur Développeur d'Applications. Même si je n'ai pas travaillé en binôme ou au sein d'une équipe, j'ai veillé à adopter une démarche professionnelle dès le départ.

J'ai fait comme si je devais **transmettre ce projet à un collègue ou à un futur développeur**. Cela m'a poussé à structurer correctement le code, à nommer les fichiers et fonctions de manière explicite, et à écrire du code lisible et cohérent. Ce choix m'a également conduit à séparer clairement les différentes couches de l'application (routes, contrôleurs, middlewares, modèles), ce qui est indispensable dans une logique de travail collaboratif.

De plus, même en solo, je me suis appuyé sur des **ressources externes** comme les documentations officielles (Express, bcrypt, JWT) ou des forums comme Stack Overflow, pour valider mes choix et résoudre des blocages. Cela m'a permis d'enrichir ma réflexion et de confronter mes méthodes à celles utilisées en entreprise.

5.2 Documentation

J'ai accordé une grande importance à la **documentation du projet**, même s'il s'agit d'un prototype personnel. J'ai rédigé un fichier `README.md` clair et structuré, décrivant l'installation, la structure de l'API, les scripts de démarrage, et les routes principales avec leurs paramètres. Mon but était de permettre à n'importe quel développeur de **prendre en main le projet en moins de 10 minutes**.

J'ai également utilisé **Postman** pour tester et documenter chaque endpoint de l'API. J'ai sauvegardé une collection contenant les différentes requêtes

(authentification, mise à jour, suppression, etc.) et les réponses attendues. Cette collection est exportable et peut facilement être utilisée par d'autres développeurs ou intégrée à un front-end React, comme je le prévois.

Enfin, tout au long du code, j'ai pris soin de **commenter les parties sensibles ou complexes**, notamment dans les middlewares ou la gestion des tokens. Cela me permet à la fois de retrouver rapidement le sens d'une fonction, et d'aider d'éventuels collaborateurs à comprendre la logique métier mise en place.

6. Bilan et retour d’expérience

6.1 Apports techniques

Ce projet a été pour moi **extrêmement formateur** sur de nombreux plans. D’un point de vue technique, il m’a permis de consolider mes compétences en Node.js, mais aussi d’approfondir des notions essentielles en **architecture backend**, notamment l’organisation d’un projet selon le modèle MVC, la création d’une API RESTful cohérente, ou encore l’utilisation des middlewares pour isoler la logique de sécurité.

J’ai également appris à **gérer des sessions de manière sécurisée** à l’aide de JSON Web Tokens (JWT), à utiliser `bcrypt` pour hasher les mots de passe, et à stocker les variables sensibles via `dotenv`. Ces pratiques sont aujourd’hui indispensables pour toute application web un tant soit peu sérieuse.

Au-delà du code, j’ai renforcé ma rigueur dans la mise en place de **tests**, avec Postman pour les tests fonctionnels, et Jest pour valider la robustesse de certaines fonctions critiques. Ce niveau d’exigence technique m’a permis de livrer une API plus fiable, plus claire et plus facilement évolutive.

6.2 Difficultés rencontrées et solutions

La première difficulté que j’ai rencontrée concernait l’**authentification via JWT**. Je comprenais le principe théorique, mais sa mise en œuvre concrète m’a obligé à revoir la manière dont les tokens sont générés, stockés côté client (dans les cookies), et surtout validés à chaque requête. J’ai dû expérimenter plusieurs approches avant de trouver un fonctionnement à la fois sécurisé et fonctionnel.

Une autre difficulté a concerné la **gestion des erreurs**. Il ne suffit pas de renvoyer un message d’erreur générique : il faut proposer des messages compréhensibles, sans pour autant divulguer d’informations sensibles. J’ai

donc mis en place une gestion centralisée des erreurs, avec des statuts HTTP cohérents, ce qui a rendu le débogage beaucoup plus simple.

Enfin, le développement en local m'a confronté à des soucis de **synchronisation entre le frontend et le backend**, notamment à cause du CORS et de la gestion des sessions. Pour y remédier, j'ai dû configurer le middleware `cors` correctement, en autorisant les bons headers et les cookies sécurisés (`credentials: true`).

6.3 Recommandations et évolutions possibles

Si je devais faire évoluer ce projet, plusieurs pistes me paraissent évidentes. La première serait d'y ajouter une **interface frontend complète** (par exemple en React), permettant à l'utilisateur de gérer son compte, consulter ses informations ou encore interagir avec d'autres utilisateurs. Cela donnerait une dimension beaucoup plus concrète à l'API.

Je pourrais également ajouter un **système de rôles plus granulaire**, permettant par exemple à un administrateur de consulter l'ensemble des utilisateurs, de supprimer un compte ou de gérer des ressources supplémentaires. Cela nécessiterait un contrôle d'accès plus poussé, mais renforcerait considérablement la pertinence du projet.

Enfin, je recommanderais d'intégrer des **logs d'activité** (connexions, modifications de données, suppressions de compte), afin de mieux tracer les actions critiques, notamment dans une optique de mise en production professionnelle.

6.4 Auto-évaluation

Je suis globalement **satisfait du travail accompli**. Le projet est fonctionnel, sécurisé et bien structuré. J'ai su mener à bien l'ensemble des objectifs que je m'étais fixés, tout en intégrant de bonnes pratiques de développement.

Je suis également conscient que certaines optimisations sont encore possibles, notamment en matière de tests automatisés, de modularisation du code, ou d'évolutivité. Mais dans le cadre de ce projet CDA, je pense avoir démontré ma capacité à **concevoir, développer et sécuriser une API** de manière autonome.

Ce projet m'a donné encore plus envie de continuer à me spécialiser dans le développement backend, et m'a montré à quel point une API bien pensée peut devenir la colonne vertébrale d'une application web moderne.