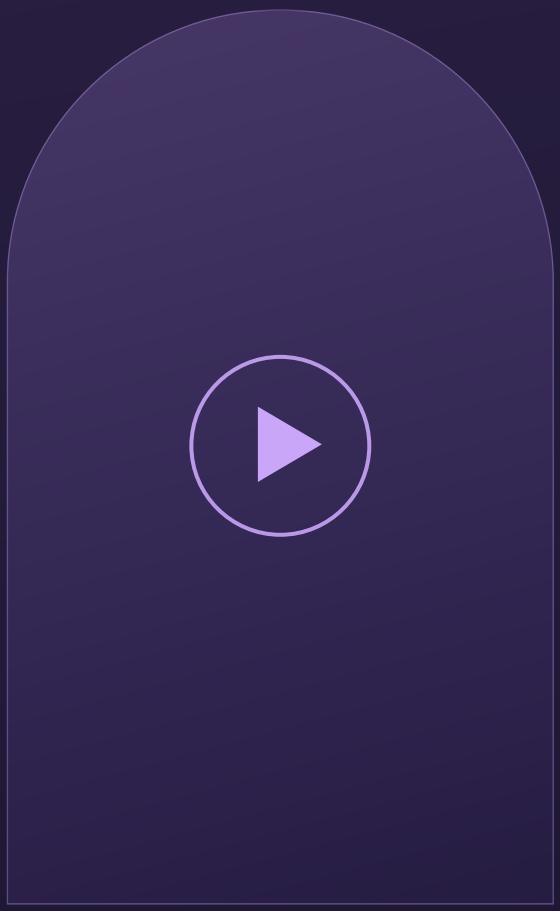
Youcus

Dossier de projet

Apprendre sur YouTube sans distraction



PRÉSENTÉ LE	27 août 2026
PRÉPARÉ PAR	RUKUNDO Ronaldo
TITRE VISÉ	Concepteur Développeur d'Applications, RNCP 37873, niveau 6
ÉCOLE	iSCOD, Visiplus Academy, session d'août 2026

Déclaration d'usage de l'intelligence artificielle

Conformément à la charte d'utilisation de l'IA générative de Visiplus Academy et de l'iSCOD, je déclare avoir utilisé un outil d'IA générative, Claude d'Anthropic, sur ce projet, à deux titres.

Sur ce dossier. Structuration du document, transcription de mes réponses orales et écrites, mise en forme, et vérification de cohérence contre les sources du projet : dépôt Git, Jira, relevés de mesures.

Sur le code de Youcus. Génération et modification de code sous ma direction, ainsi que relecture critique. J'ai défini l'architecture, les choix techniques et le découpage en tickets ; j'ai dit quoi faire et dans quel ordre ; j'ai relu les modifications, corrigé les orientations qui partaient dans le mauvais sens, et validé chaque intégration. Tous les commits du dépôt sont signés de ma main, et chaque livraison est passée par mon test manuel avant sa fusion.

Le motif. Le chantier d'infrastructure qui devait couvrir mes compétences de déploiement dans le cadre de mon alternance a été écarté au profit d'autres priorités de l'entreprise (section 3). Youcus, un projet que je portais déjà pour mon propre usage, est devenu le seul terrain possible pour ces compétences, et la fenêtre avant le dépôt s'était déjà refermée. J'ai donc fait le choix, sous contrainte de calendrier, de m'outiller pour mener seul un projet complet de la conception au déploiement, plutôt que de présenter un dossier amputé.

Ce choix d'outillage n'a pas déplacé le rôle que j'ai tenu : celui d'un architecte qui conçoit, décide, dirige et vérifie. J'ai testé chaque ticket à la main avant de le fermer, et j'ai arrêté le travail quand l'implémentation s'écartait de mes maquettes pour la faire reprendre. La section 4 rend compte de la méthode, les sections 9 et 14 des décisions techniques que j'ai prises et de celles sur lesquelles je suis revenu.

Les décisions, l'architecture, les arbitrages et les analyses présentés dans ce dossier sont les miens, et je peux en défendre chaque ligne à l'oral. Les échanges avec l'outil sont conservés et peuvent être communiqués sur demande.

Sommaire

1. Liste des compétences mises en œuvre

Les onze compétences professionnelles du référentiel (REV2 CDA) sont toutes mises en œuvre dans ce projet.

#	COMPÉTENCE	PREUVE PRINCIPALE	OÙ	COUVERTURE
1	Installer et configurer son environnement de travail	Poste Ubuntu, Docker Compose, VS Code, Postman, dépôt et CI	section 3, section 13	Complète
2	Développer des interfaces utilisateur	Lecteur focus et éditeur de notes, du wireframe à l'écran livré	section 8, section 15	Complète, hors audit d'accessibilité (section 15)
3	Développer des composants métier	Synchronisation d'une playlist, export RGPD, logique de progression	section 16	Complète
4	Contribuer à la gestion d'un projet informatique	Backlog Jira, sprints, Gantt et vélocité, une PR par ticket	section 4	Complète
5	Analyser les besoins et maquetter une application	Expression des besoins, matrice besoin/spécification/ticket, Balsamiq puis Figma	section 2, section 5, section 8	Complète
6	Définir l'architecture logicielle	Front statique et API découplée, justifiée puis vérifiée sur facture	section 7	Complète
7	Concevoir et mettre en place une base de données relationnelle	MCD, MLD, refactor en N:N et migration écrite à la main	section 9, section 10, annexe E	Complète, contrainte des notes portée par le code (section 9)
8	Développer des composants d'accès aux données SQL et NoSQL	Prisma sur MySQL, cache Redis en lecture-écriture différée	section 17	Complète
9	Préparer et exécuter les plans de tests	Stratégie de test argumentée, couverture mesurée, jeu d'essai réel	section 20, section 21	Complète, sans test de bout en bout navigateur (section 20)
10	Préparer et documenter le déploiement	Conteneurisation, migrations au démarrage, livrables et documentation d'API	section 6, section 13	Complète, sans procédure de retour arrière formalisée (section 23)
11	Contribuer à la mise en production dans une démarche DevOps	Chaîne GitHub Actions, branche protégée, déploiement automatique, run rouge puis vert	section 4, section 20	Complète

La colonne « couverture » signale, le cas échéant, ce qui n'est pas couvert et renvoie à la section qui l'explique. Les pages se lisent au sommaire : je préfère un renvoi de section, qui

reste juste, à un numéro de page qui cesserait de l’être à la moindre modification du document.

2. Expression des besoins

L'idée de Youcus vient d'un constat personnel : YouTube héberge une quantité énorme de contenu éducatif de qualité (des formations entières sont disponibles en playlists) mais la plateforme elle-même est remplie de distractions. Recommandations, autoplay, commentaires : il est difficile d'y rester concentré, parce que tout y est conçu pour capter l'attention plutôt que pour la protéger.

Ce constat, je le fais en payeur. J'ai acheté des cours sur Udemy alors que des contenus de niveau équivalent sont librement accessibles sur YouTube : **freeCodeCamp** y publie des formations complètes de plusieurs heures, **Traversy Media**, **Fireship**, **JavaScript Mastery** ou **NetworkChuck** couvrent le développement web et les réseaux avec une exigence professionnelle. Ce que j'achetais chez Udemy n'était donc pas le savoir, c'était le cadre : une progression, une trace de ce qui a été vu, un endroit où prendre des notes.

J'ai donc voulu créer une plateforme gratuite pour s'auto-former tranquillement, en combinant le meilleur des deux mondes : le contenu libre et gratuit de YouTube, et l'expérience structurée d'Udemy, dont je me suis inspiré pour le design, un cours = une playlist, une progression visible, des notes à côté de la vidéo. On peut y importer ses playlists éducatives, y compris celles qu'on a soi-même composées sur YouTube.

Étant à la fois le concepteur et le premier utilisateur, j'ai formalisé ce besoin comme je l'aurais recueilli auprès d'un client, en langage non technique : « je veux disposer d'un espace pour suivre mes playlists de formation, sans distraction, en gardant mes notes et ma progression au même endroit ».

De cette réflexion sont sortis cinq besoins : se connecter avec son compte Google et importer ses playlists YouTube ; regarder les vidéos dans un lecteur épuré, sans recommandations ; prendre des notes attachées à chaque vidéo et à chaque playlist ; suivre sa progression ; et garder la maîtrise de ses données personnelles (export, suppression).

Ces besoins ont été posés avec leurs limites. Youcus ne remplace pas YouTube : les conditions d'utilisation de l'API imposent le lecteur officiel, interdisent tout téléchargement et interdisent de toucher à la publicité (section 6). L'abonnement payant, présent dans le périmètre initial, a été retiré en cours de projet pour tenir le calendrier. Enfin, l'application est hébergée sous l'adresse fournie par la plateforme de déploiement, sans nom de domaine propre, s'agissant d'un projet de formation.

3. Présentation de l’entreprise et du service

Cette section présente deux choses distinctes, et la distinction compte : l’entreprise où je suis en alternance, puis le projet sur lequel porte ce dossier. Youcus n’est pas un projet d’entreprise. Néatemys n’y a pris aucune part.

L’entreprise où je suis en alternance

Je suis en alternance chez Néatemys, à Lyon.

Néatemys est la marque de MBUSINESSEUROPE, société par actions simplifiée créée fin 2017 et dont le code d’activité est l’édition de logiciels applicatifs. La marque Néatemys elle-même a été déposée début 2021. L’entreprise est également déclarée comme organisme de formation.

Son marché est celui de la résolution des conflits en entreprise. Néatemys est un centre de médiation certifié par le ministère de la Justice, et son offre repose sur un processus qui associe un médiateur expérimenté à un expert du domaine concerné, avocat, consultant ou coach. Ce qui est vendu n’est donc pas le service de médiation seul, mais la plateforme qui l’outille : messagerie et visioconférence sécurisées, suivi du dossier en temps réel, assistant de rédaction des demandes et des accords. Les clients sont des entreprises et des associations, le cas le plus fréquent étant le conflit entre associés.

L’entreprise exploite deux plateformes distinctes, et c’est sur les deux que je travaille. Néatemys porte la médiation : c’est la plateforme décrite ci-dessus, celle qui outille le processus médiateur et expert. NEA-RH s’adresse aux directions des ressources humaines et propose des services d’accompagnement des salariés, dont l’assistance psychologique.

Le service et mon poste

L’équipe salariée compte trois personnes : la directrice, une développeuse senior et moi. Les intervenants qui traitent les dossiers ne sont pas salariés : les médiateurs et les experts côté Néatemys, les psychologues côté NEA-RH, travaillent tous en prestation, et c’est la directrice qui les sélectionne elle-même. C’est ce modèle qui permet à une équipe de trois personnes de faire tourner deux plateformes.

Il n’y a donc pas de service informatique distinct du reste de l’entreprise : le développement, c’est la développeuse senior et moi, et la directrice fixe les priorités fonctionnelles. Cette structure a une conséquence directe sur ma façon de travailler, et c’est elle qui rend la section 4 lisible : chaque décision technique se prend à deux, chaque livraison est relue par la seule autre personne qui code.

Je suis développeur fullstack chez Néatemys depuis août 2023, soit près de trois ans, dont onze mois sous statut d’alternant iSCOD depuis le 1er septembre 2025. J’y travaille quotidiennement en Django, HTML, JavaScript et CSS/Tailwind, sur un code versionné dans Bitbucket, au sein d’une organisation Scrum.

Mon travail couvre les deux plateformes, de l’interface au serveur. Côté interface, j’ai mené la refonte des composants et leur passage à Tailwind. Côté serveur, j’écris des endpoints métier, j’interviens sur la couche temps réel qui porte la visioconférence, et je traite les migrations de données.

Chaque livraison suit la même chaîne : je développe en local, je déploie en préproduction pour la tester, j'ouvre une pull request que la développeuse senior relit, puis la directrice valide fonctionnellement avant la mise en production. C'est cette chaîne que j'ai cherché à reproduire seul sur Youcus, et la section 4 dit ce qui s'y transpose et ce qui ne s'y transpose pas.

Le projet Youcus, et ce qui l'en sépare

Youcus est un projet personnel, conçu et développé seul, en dehors du cadre de mon entreprise et sur mon temps propre. Aucun code, aucune donnée et aucun outil de Néatemys n'y ont été utilisés, et réciproquement.

Le calendrier de ce projet, lui, n'était pas le plan de départ. Il était prévu que je couvre les compétences de déploiement et de mise en production dans le cadre de mon alternance : la mise en place d'une chaîne d'intégration continue avec Jenkins et la conteneurisation de l'application avec Docker figuraient au programme de l'année. D'autres priorités sont passées devant, et ce chantier n'a pas eu lieu. C'est aussi la raison pour laquelle je travaille encore sans conteneurs en entreprise. Et c'est ce qui a imposé le calendrier de Youcus : quand il est devenu clair que ces compétences ne seraient pas couvertes par mon poste, la fenêtre avant le dépôt s'était déjà refermée, et le projet a dû être mené dans l'urgence. Plutôt que de présenter un dossier amputé de ces compétences, j'ai construit Youcus pour les exercer réellement : définir une architecture, concevoir et faire évoluer une base de données, préparer et documenter un déploiement, puis tenir une chaîne d'intégration et de livraison continues jusqu'en production.

Une précision s'impose ici, parce qu'elle change la lecture de ce dossier. Youcus n'est pas un projet de substitution, monté faute de mieux. C'est une idée que je portais déjà, née de mon propre usage : j'apprends beaucoup sur YouTube, quotidiennement, et j'en connais les défauts par expérience plutôt que par observation (section 2). Ce que le changement de priorités a décidé, ce n'est donc pas de faire ce projet, c'est de le faire maintenant et vite.

Ce que j'ai transposé de mon équipe, ce sont ses méthodes de travail, et elles seules : tickets estimés, branche par ticket, pull request, revue de code, intégration continue. La section 4 rend compte de cette transposition et de ses limites, puisque la revue par un tiers, elle, ne se transpose pas quand on travaille seul.

L'environnement technique, en revanche, n'a rien de commun avec celui de l'entreprise. Chez Néatemys je travaille en Django, sans conteneurs, sur Bitbucket. Youcus est en TypeScript, entièrement conteneurisé, sur GitHub. Ce sont deux mondes séparés, et c'est le second que décrit ce dossier.

Youcus a été développé sur ma machine personnelle, un ordinateur portable sous Ubuntu Linux, avec des outils que j'ai choisis seul. Je développe dans VS Code avec un terminal, et je fais tourner l'environnement complet du projet en local (base MySQL, cache Redis, API et front) grâce à Docker et docker-compose, l'ensemble démarrant d'une seule commande. Cette conteneurisation est un choix que j'ai fait pour ce projet, et que j'ai dû apprendre pour lui.

Je teste dans Chrome et dans Firefox, dont les outils de développement (en particulier les onglets Network et Console) ont joué un rôle direct dans le diagnostic des incidents racontés à la section 14 ; Postman m'a servi à éprouver l'API indépendamment du front. La conception des interfaces s'est faite sur Balsamiq pour la basse fidélité, puis Figma pour la haute fidélité. La gestion de projet passe par Jira et le code par GitHub. Travaillant seul sur ce projet, je n'avais pas d'outil de communication d'équipe à y associer.

4. Gestion de projet

Méthode et outillage

Pour gérer Youcus, j'ai appliqué la méthode que je pratique quotidiennement chez Néatemys, où je travaille en alternance comme développeur fullstack (Django, HTML, JavaScript, CSS/Tailwind). Nous y fonctionnons en Scrum à trois : la directrice fixe les priorités fonctionnelles, puis la développeuse senior et moi découpons le besoin en tickets estimés que nous nous répartissons dans des sprints d'une semaine. Chaque ticket suit le même circuit : branche dédiée, pull request, revue de code, la développeuse senior relit toutes mes PRs et c'est elle qui merge ; je relis de mon côté une partie des siennes. Le code vit sur Bitbucket, la communication passe par Slack.

Sur Youcus, seul, j'ai gardé de ce fonctionnement ce qui a du sens en solo : un backlog Jira structuré en epics et user stories estimées (62 tickets : 10 epics, 46 stories, 6 tasks), des sprints courts, une branche et une pull request par ticket.

J'ai en revanche perdu la revue de code par un tiers : c'est une limite du travail en solo que j'assume : la CI (lint, typecheck, tests sur chaque PR) et la branche `main` protégée servaient de seul garde-fou avant merge. Conformément à la charte de mon école, je précise que j'ai outillé cette gestion de projet avec un assistant IA connecté à Jira, notamment pour la création des tickets en lot lors des sessions de cadrage.

Le déroulé réel

Le projet s'est déroulé en trois phases au rythme dicté par mon alternance. La première est celle de la conception : début juin, avant d'écrire la moindre ligne de code, j'ai produit le cahier des charges et l'ensemble des modèles du projet (cas d'utilisation, données, classes, séquences, architecture, activité et états) soit onze diagrammes bouclés le 7 juin. Le 12, une session de cadrage a traduit cette conception en backlog : j'ai construit dans Jira les 8 epics initiaux et les premières user stories, avec critères d'acceptation et estimations. De fin juin à mi-juillet, la conception s'est étalée en parallèle d'une charge importante chez Néatemys : design system, wireframes basse fidélité puis maquettes haute fidélité des 7 écrans sur Figma, et le sprint 1 (socle technique) resté ouvert pendant cette période. Mi-juillet, le chantier d'infrastructure qui devait couvrir mes compétences de déploiement en entreprise a été écarté au profit d'autres priorités (section 3). Youcus devenait donc le seul terrain possible pour ces compétences, et il ne restait que quelques semaines. J'ai fait le choix d'accélérer plutôt que de réduire le périmètre : les 18, 19 et 20 juillet, j'ai exécuté l'essentiel du développement en trois jours pleins, organisés en micro-sprints thématiques ouverts et clos dans la journée (socle et CI/CD le premier jour ; authentification, playlists et lecteur le deuxième, avec le premier déploiement en production ; notes, RGPD et tests le troisième). Cette exécution compressée n'a été possible que parce que la conception était terminée : maquettes prêtes, backlog découpé, architecture décidée.

Le diagramme de Gantt et le velocity chart (générés depuis les données réelles de Jira) montrent ce découpage : un long sprint de socle, puis cinq micro-sprints de 3 à 9 tickets.

Versionnement et qualité

Le dépôt GitHub applique les règles que je connais du travail en équipe : branche `main` protégée (aucun push direct, passage obligatoire par pull request), commits au format

Conventional Commits référençant leur ticket (81 commits sur `main` et 51 pull requests, dont 36 fusionnées : les 15 restantes sont toutes des montées de version proposées par Dependabot, 6 encore ouvertes et 9 refermées au profit de versions plus récentes, traitées à la section 22), et une CI GitHub Actions qui exécute lint, typecheck, tests et build sur chaque PR, doublée d'une CD qui build et déploie automatiquement sur Sevalla à chaque merge vers `main`.

Cette CI m'a concrètement arrêté : le 19 juillet, en plein développement de l'import de playlists, le job est passé au rouge en 28 secondes, mes tests de `fetchPlaylist` dépendaient de la clé YouTube présente dans mon `.env` local, absente de l'environnement CI, et le code renvoyait une erreur 503 là où le test attendait un 404. J'ai corrigé en rendant les tests indépendants de l'environnement (mock du fetch), une leçon directe sur la reproductibilité des tests. Travaillant seul, sans relecteur humain, ce filet automatisé était ma seule barrière avant la production : il a joué son rôle.

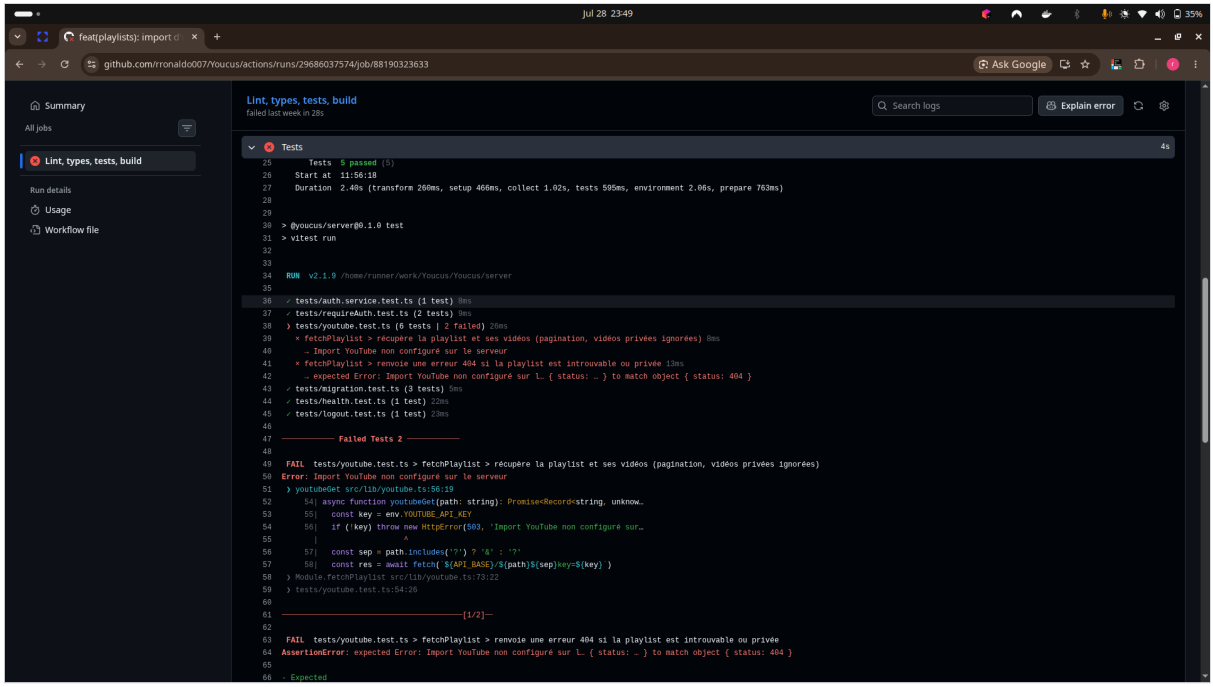


Figure 1. *GitHub Actions : le run rouge du 19/07 : le job « Lint, types, tests, build » échoue en 28 s, sur deux tests de `fetchPlaylist`*

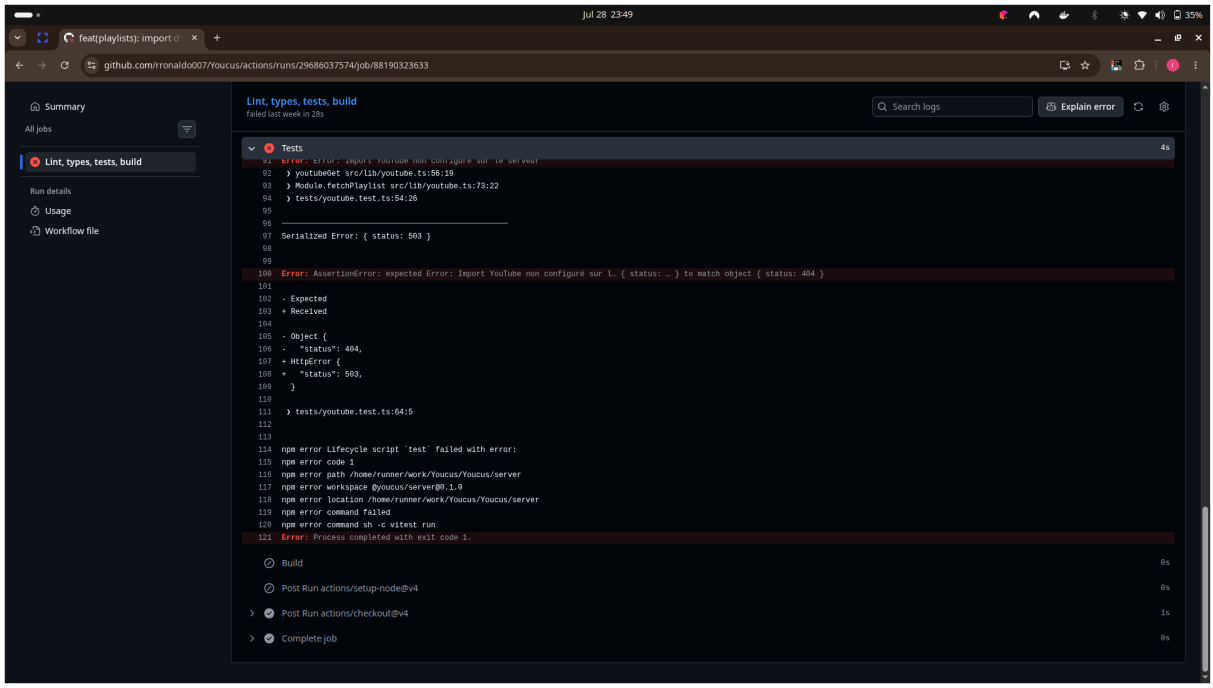


Figure 2. *Le détail de l'échec : le test attendait `{ status: 404 }` et a reçu `{ status: 503 }`, faute de clé YouTube dans l'environnement d'intégration*

Arbitrages et bilan

Le backlog n'est pas resté figé, et les deux epics ajoutés en cours de route racontent la même prise de conscience. Le 23 juin, en attaquant les maquettes, j'ai créé l'epic « Design et interface utilisateur » (CS-24), absent du cadrage initial : le travail de conception a révélé que l'interface méritait un chantier à part entière, avec sa bibliothèque de composants. Le 19 juillet, au moment de coder cette interface, j'en ai ouvert un dixième, « UI / Design System » (CS-52), pour porter l'implémentation écran par écran. Concevoir une interface et la construire sont deux travaux distincts, avec des livrables et un rythme différents ; les mélanger dans un seul epic aurait rendu le suivi illisible. À l'inverse, l'epic premium/paiement, présent depuis le premier jour, est volontairement resté au backlog : il n'apporte rien à la valeur cœur du produit (apprendre sans distraction) et aucune de ses fonctionnalités n'était nécessaire pour une première version complète. Prioriser, c'est aussi décider de ce qu'on ne fait pas.

Ce que je retiens de cette gestion de projet en solo : la méthode d'équipe se transpose, mais pas intégralement, les tickets, les sprints et la CI gardent tout leur sens seul ; le daily et la revue par un tiers, non. Mon vrai enseignement est le rapport entre préparation et exécution : c'est parce que la conception était finie que trois jours ont suffi à développer l'essentiel. Et la limite est claire : sans relecteur, la qualité repose entièrement sur l'outillage automatisé.

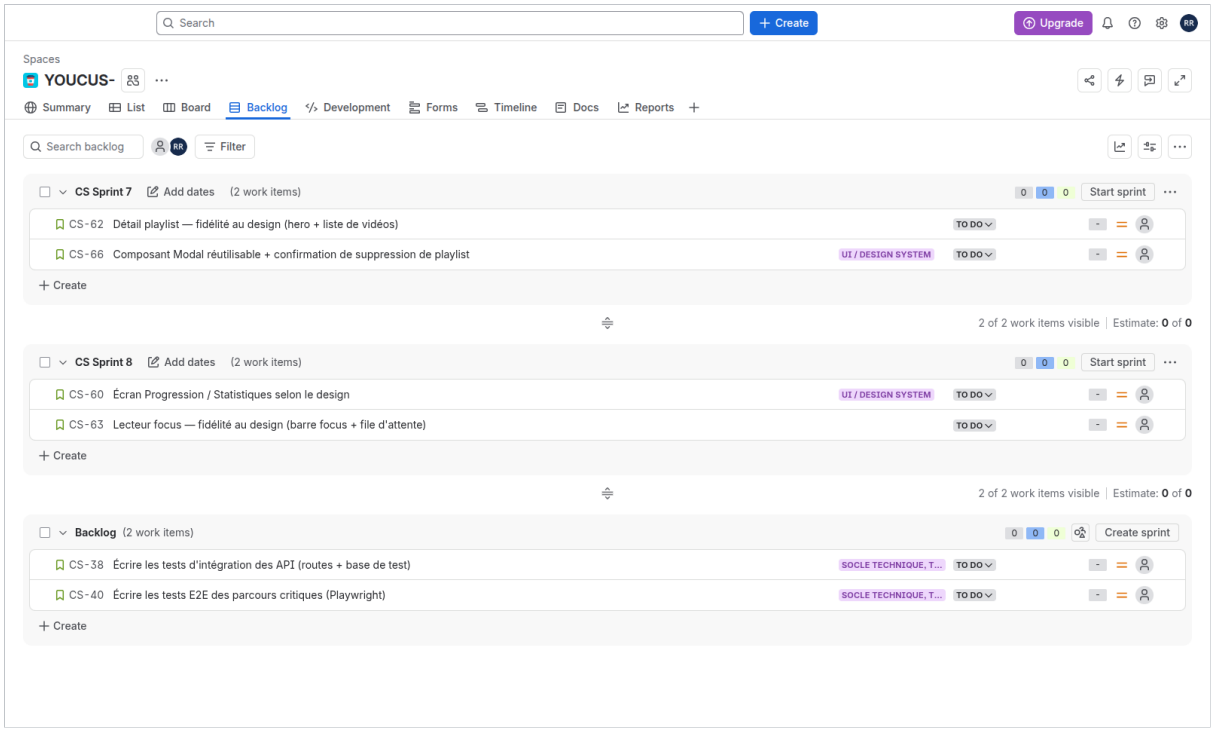


Figure 3. *Jira : le backlog organisé en sprints*

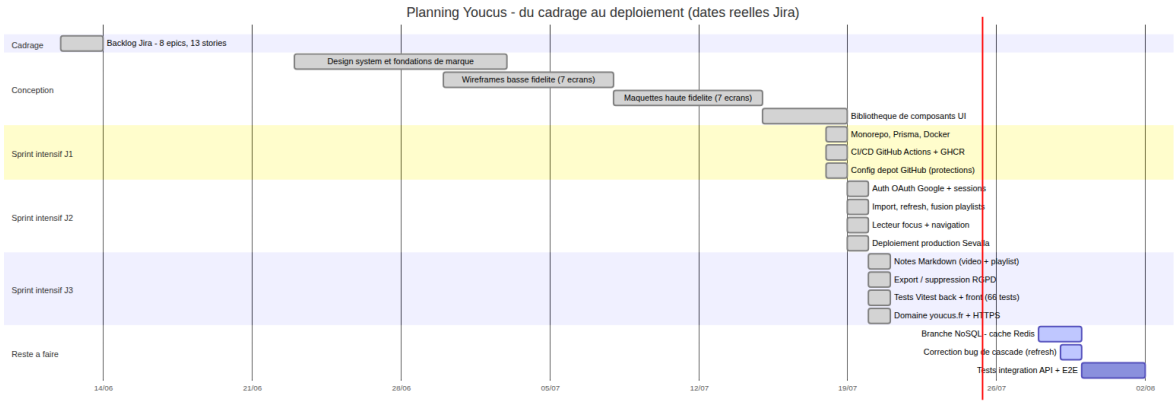


Figure 4. *Diagramme de Gantt (généré depuis les dates réelles des tickets Jira)*

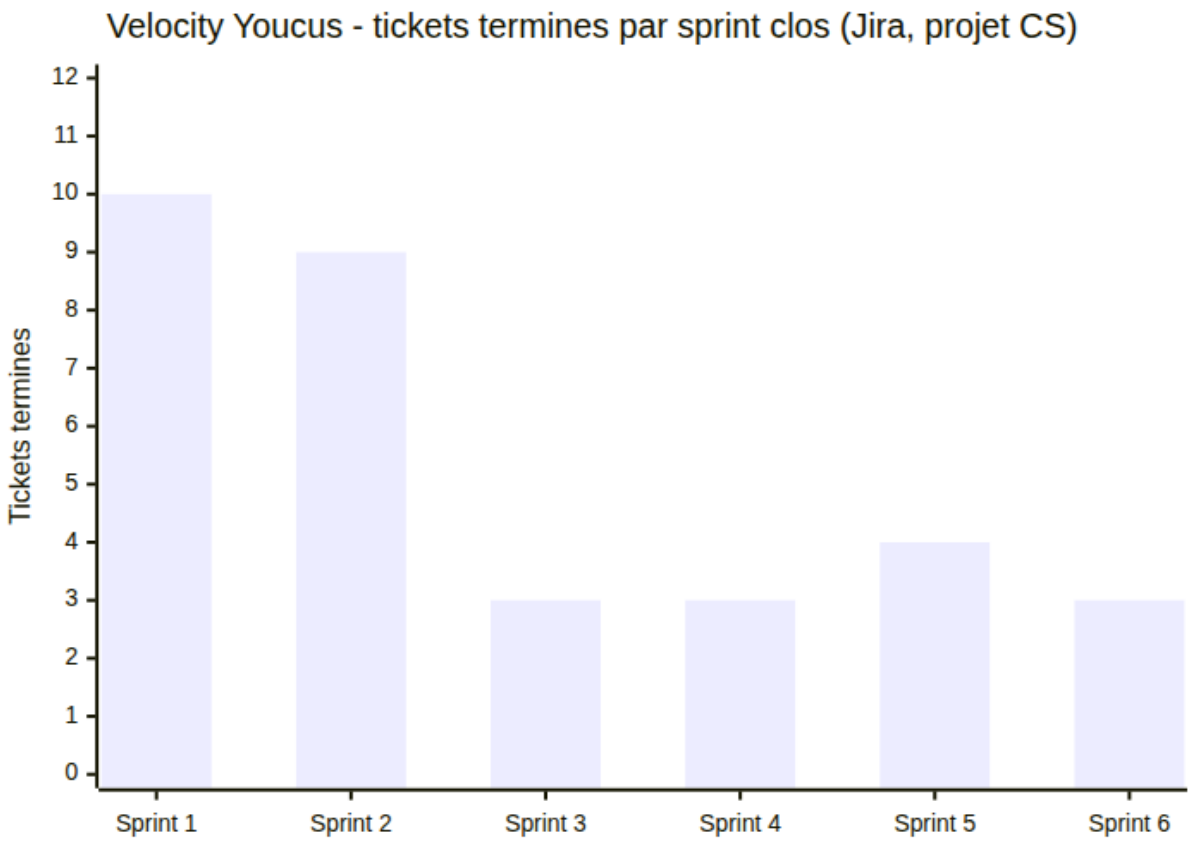


Figure 5. *Vélocité par sprint (6 sprints clos)*

5. Spécifications fonctionnelles

Les cinq besoins exprimés à la section 2 ont été traduits en spécifications techniques. Le tableau ci-dessous est une **matrice de traçabilité** : chaque ligne relie un besoin formulé en langage d'utilisateur à la solution technique retenue, et au ticket Jira qui l'a portée. C'est ce qui garantit qu'aucun besoin n'a été perdu en route, et qu'aucune fonctionnalité n'a été développée sans besoin derrière.

BESOIN (SECTION 2)	SPÉCIFICATION FONCTIONNELLE	TICKETS
Se connecter avec son compte Google et importer ses playlists	Authentification **OAuth 2.0** déléguée à Google, portée `youtube.readonly`, session portée par un cookie signé. Import via la **YouTube Data API v3**, par URL publique ou depuis ses propres playlists, avec rafraîchissement à la demande.	CS-9, CS-11, CS-13, CS-54, CS-58
Regarder les vidéos sans distraction	**IFrame Player API** paramétrée (`rel=0`, `modestbranding`, annotations désactivées), navigation entre les vidéos d'une playlist par une liste latérale.	CS-14, CS-15
Prendre des notes attachées à la vidéo et à la playlist	Éditeur **Markdown** avec sauvegarde automatique, aperçu formaté et **assaini** avant rendu. Une note par utilisateur et par cible.	CS-16, CS-17, CS-49
Suivre sa progression	Position de lecture remontée pendant la lecture effective, reprise à la position enregistrée, marquage manuel d'une vidéo comme vue.	CS-18, CS-19
Garder la maîtrise de ses données	**Export** de toutes les données personnelles au format JSON et **suppression** du compte, avec effacement en cascade.	CS-22

Deux spécifications ne découlent d'aucun besoin utilisateur mais d'une contrainte : la **fusion de playlists** (CS-55), née de l'usage réel (plusieurs playlists couvrant le même sujet) et le **cache des métadonnées YouTube** (CS-67), imposé par le quota de l'API et détaillé à la section 17.

6. Contraintes et livrables attendus

Contraintes fonctionnelles : ce que les CGU de YouTube interdisent

La contrainte structurante du projet n'est pas technique, elle est **contractuelle**. Youcus se construit sur une API tierce dont les conditions d'utilisation encadrent strictement ce qu'on peut en faire :

- **Aucun téléchargement de vidéo.** Le contenu ne peut être ni copié, ni stocké, ni rediffusé. Youcus ne conserve que des métadonnées : titres, miniatures, identifiants.
- **Le lecteur officiel est obligatoire.** La lecture doit passer par le player YouTube ; il n'est pas permis de construire son propre lecteur au-dessus du flux.
- **Les publicités ne peuvent être ni bloquées, ni modifiées, ni remplacées.** C'est explicite dans les *YouTube API Services Terms of Service*, et cela a redéfini la promesse du produit : Youcus supprime la dérive, pas la publicité (voir section 14).

Ces trois contraintes ont été lues **avant** de concevoir, pas découvertes en cours de route. Elles expliquent pourquoi l'architecture est celle d'une **couche d'étude posée sur YouTube** plutôt que d'un concurrent de YouTube.

Contraintes non fonctionnelles

CONTRAINTE	NATURE	RÉPONSE APPORTÉE
Quota de la YouTube Data API : 10 000 unités/jour	Externe, non négociable	Cache Redis en *cache-aside* sur les métadonnées (section 17), et calcul d'exposition documenté
RGPD : données personnelles d'utilisateurs européens	Légale	Export et suppression implémentés, effacement en cascade au niveau du schéma (section 10, section 19)
Sécurité : OAuth, XSS, CSRF, secrets	Technique	Détaillée intégralement à la section 19, avec ses limites assumées
Coût d'hébergement maîtrisé	Économique	Conteneurisation Docker, déploiement sur Sevalla, base MySQL unique, cache en mémoire sans persistance

La contrainte de quota mérite d'être soulignée parce qu'elle est **partagée** : le plafond vaut pour le projet entier, pas par utilisateur. Un seul usage abusif pénaliserait tout le monde jusqu'au lendemain. C'est ce qui justifie que le cache soit une décision d'architecture et non une optimisation tardive.

Une précision lève l'ambiguïté la plus fréquente sur ce point : **regarder une vidéo ne consomme aucune unité de ce quota.** Le plafond porte sur la *YouTube Data API v3*, celle qui sert à lister et décrire les contenus ; la lecture passe par l'*IFrame Player API*, un composant d'affichage qui n'est pas décompté du quota de la Data API. Le coût de Youcus se concentre donc entièrement sur l'import et le rafraîchissement.

Les deux seuls points d'appel sont `playlists.list` et `playlistItems.list`, à 1 unité par requête, avec une pagination fixée à 50 éléments : la playlist de 193 vidéos du jeu d'essai (section 21) revient ainsi à 5 unités, une pour la playlist et quatre pour ses pages. Aucun appel à `search.list`, qui coûterait à lui seul 100 unités, n'a été nécessaire. C'est ce coût

unitaire faible mais **répété à chaque consultation** qui justifie le cache : il ne protège pas d'un import isolé, il protège de la relecture des mêmes métadonnées.

Livrables attendus

1. **L'application déployée** en production, accessible publiquement (Sevalla).
2. **Le code versionné** sur GitHub, avec un historique de tickets et de pull requests traçable.
3. **La documentation** : cahier des charges, spécifications techniques, diagrammes UML et Merise, documentation d'API (OpenAPI), documentation développeur.
4. **Les tests** et leur automatisation en intégration continue (section 20).

7. Justification de l’architecture

J’ai choisi une **architecture découplée** : d’un côté un build statique React, de l’autre une API REST Express indépendante. L’alternative aurait été l’architecture intégrée que je pratique quotidiennement chez Néatemys, où Django fabrique les pages côté serveur selon le patron MVT : du **rendu côté serveur** (SSR). Youcus fonctionne à l’inverse : le serveur envoie un document vide et un bundle JavaScript, et c’est le navigateur qui construit les vues.

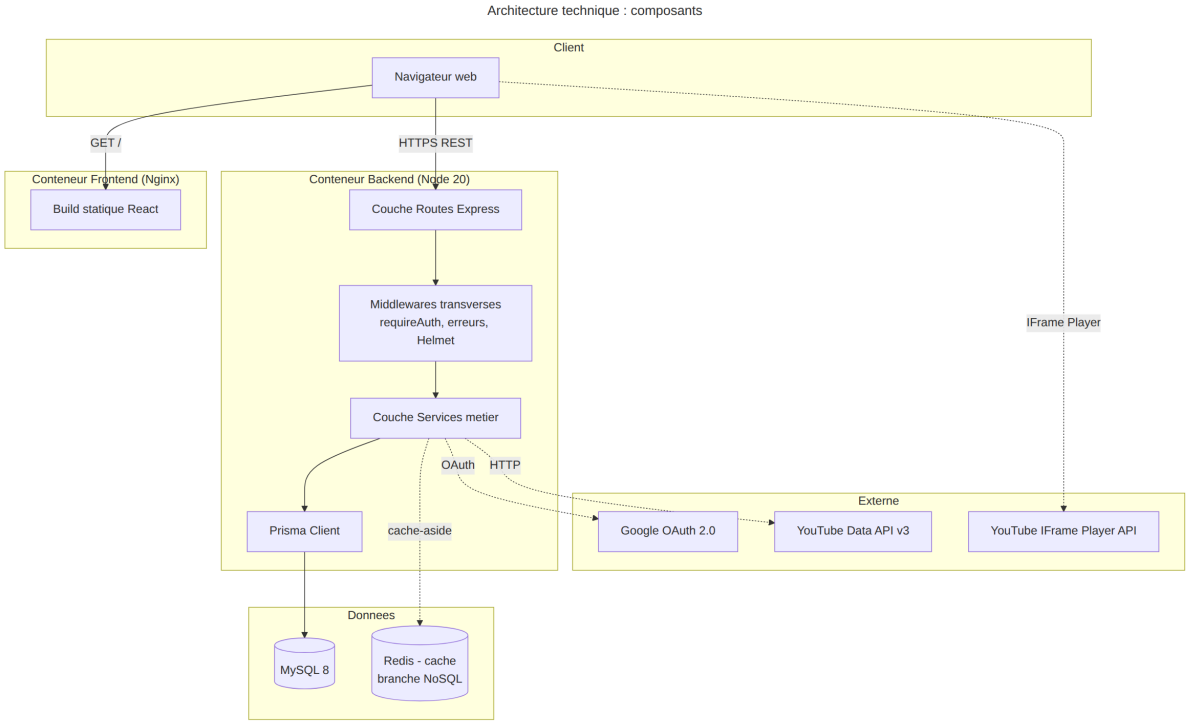


Figure 6. *Architecture des composants : client statique, API en couches, services externes (diagramme 07)*

C’est du **rendu côté client** (CSR), dans une **application à page unique** (SPA). Précision qui a son importance : le front est *servi* statiquement, mais il n’est pas *pré-rendu*, ce n’est pas de la génération de site statique (SSG), où le HTML serait calculé à la compilation. Deux raisons m’ont fait trancher pour ce découplage.

La première est l’évolution du produit. Youcus est un projet que je compte continuer, et une application mobile en fait partie. Une API qui ne connaît pas son client pourra en servir plusieurs demain (web, mobile) sans être réécrite.

La seconde est le déploiement. Un front statique se sert sans processus qui tourne : sur Sevalla, seule l’API consomme un conteneur, ce qui réduit d’autant le coût d’hébergement et la surface de panne.

Ce choix a un coût, et je l’ai appris à mes dépens. Le client et l’API vivant sur deux domaines distincts, les cookies ne se partagent pas entre eux, et la section 19 en tire les conséquences sur la session. Et le retour OAuth doit atterrir sur le domaine de l’API, pas sur celui du front : la confusion entre ces deux adresses est précisément l’incident raconté à la section 14.

Éco-conception et sobriété technique

Plusieurs de ces choix d’architecture ont un effet direct sur les ressources consommées, et je les revendique aussi à ce titre. Servir le front en fichiers statiques plutôt que par un processus supprime un serveur qui tournerait en permanence pour rendre des pages identiques. Le cache Redis évite de réinterroger l’API YouTube pour des données qui n’ont pas changé, ce qui économise à la fois du quota et des appels réseau. Youcus ne duplique aucune vidéo : il ne stocke que des métadonnées et délègue la diffusion à YouTube, là où un téléchargement local multiplierait le stockage pour un contenu déjà hébergé. La conteneurisation, enfin, dimensionne l’API au strict nécessaire.

Ces décisions ont une mesure : sur un mois complet, l’hébergement a coûté moins de quatre dollars, dont zéro pour le front statique. L’application n’ayant pas encore de charge réelle, ce n’est pas une performance environnementale ; cela montre seulement que la sobriété d’une architecture se vérifie sur une facture.

8. Prototypage

J’ai maquetté en deux temps, avec deux outils choisis pour deux intentions différentes.

D’abord Balsamiq, appris pendant ma formation. Sa basse fidélité est volontaire : elle permet d’explorer vite, et de jeter sans regret. J’y ai produit **24 planches** : les cinq écrans principaux (dashboard, lecteur focus avec notes, landing et connexion, progression, paramètres et données RGPD), leurs déclinaisons mobile et tablette, puis les états qu’on oublie toujours au début : les modales d’import simple et multi-sélection, la fusion de playlists, la connexion OAuth, les écrans de chargement en squelette, le détail des notes. Traiter l’adaptation aux écrans dès ce stade m’a évité de découvrir les problèmes de mise en page une fois le code écrit. *(Projet Balsamiq : https://balsamiq.cloud/smq3s6u/pmcoi8f)*

Ensuite Figma, pour la haute fidélité : un design system (couleurs, typographie, espacements), les 7 écrans finaux déclinés en thème clair et sombre, une bibliothèque de composants réutilisables et des pages dédiées aux états de chargement et au responsive. *(Fichier Figma : https://www.figma.com/design/seZpx035xOk2Rerl8XuS0Z/youcs)*

Ma référence visuelle assumée était Udemy. J’en ai repris trois codes qui ont fait leurs preuves sur l’apprentissage vidéo : la liste des leçons en panneau latéral à côté du lecteur, la progression visible partout, et les cartes de cours du tableau de bord : appliqués cette fois au contenu libre de YouTube.

L’enchaînement des écrans est donné en annexe C (landing vers connexion Google vers dashboard vers détail de playlist vers lecteur focus, avec les chemins vers les notes, la progression et les paramètres). Le trio ci-dessous (wireframe, maquette, écran final) montre le chemin complet du lecteur focus, de l’intention au produit.

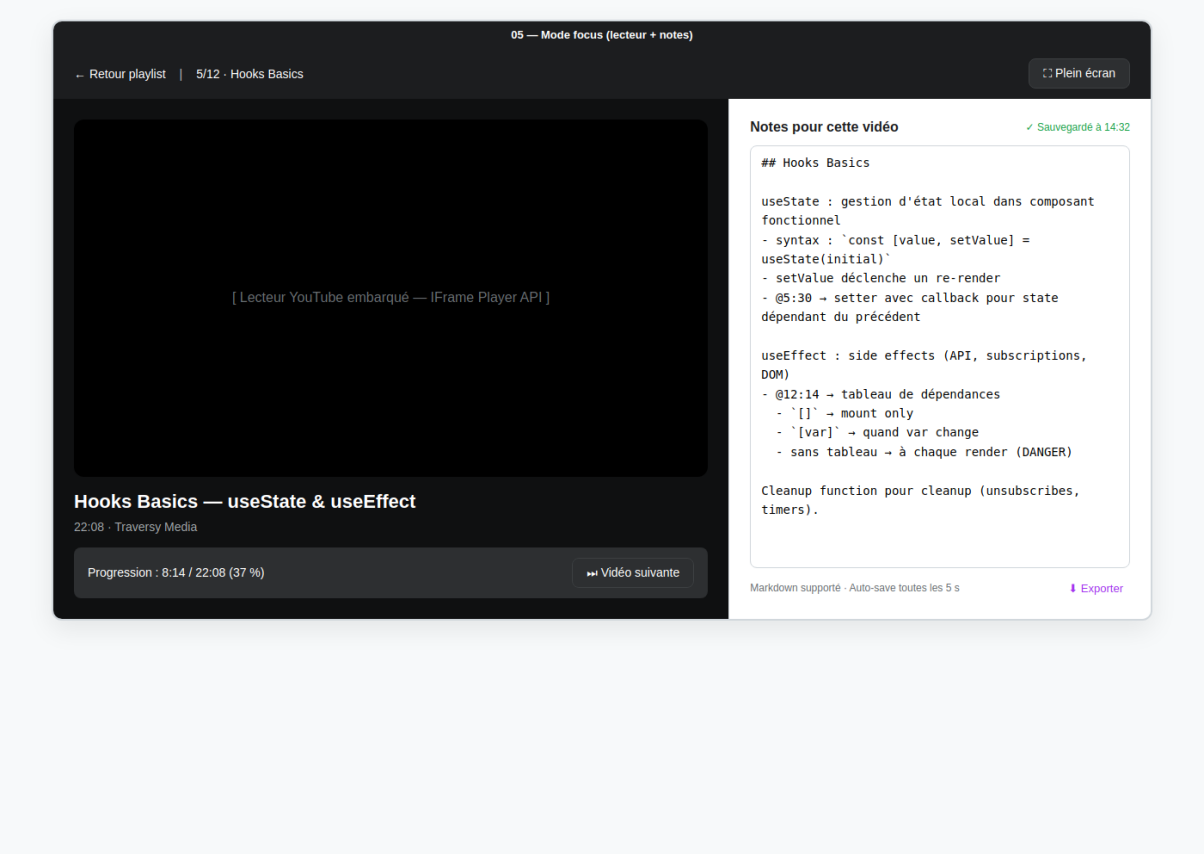


Figure 7. *Wireframe Balsamiq : le lecteur focus (basse fidélité)*

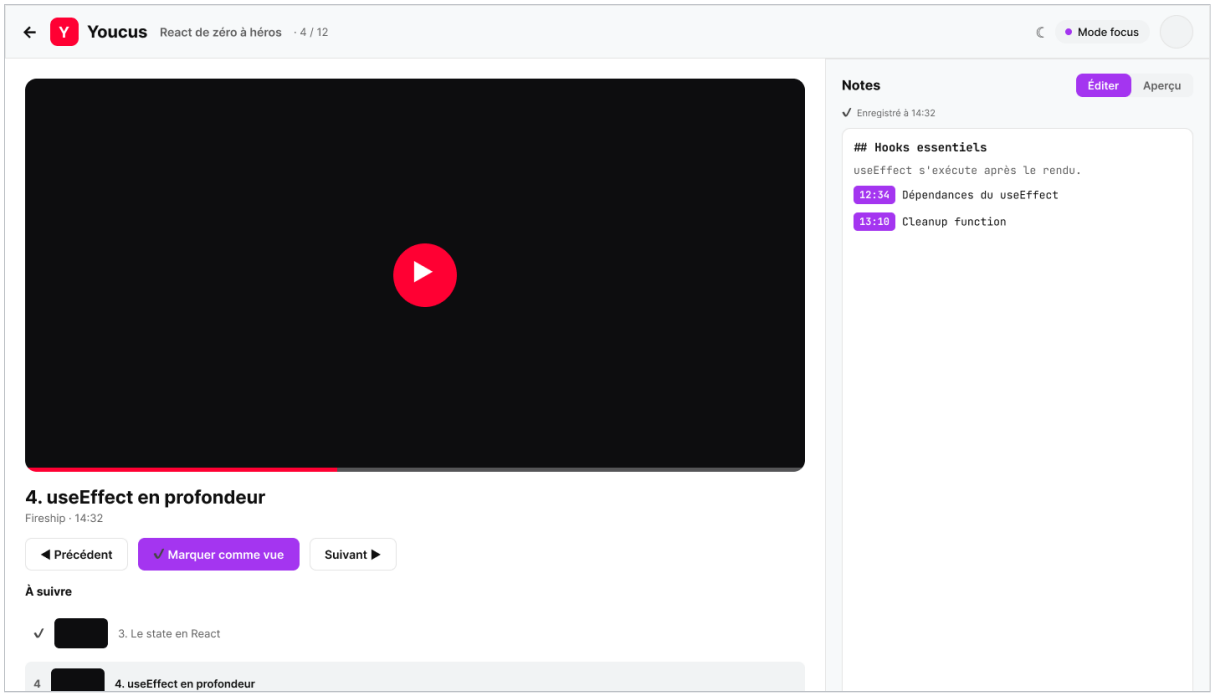


Figure 8. *Maquette Figma : le lecteur focus (haute fidélité)*

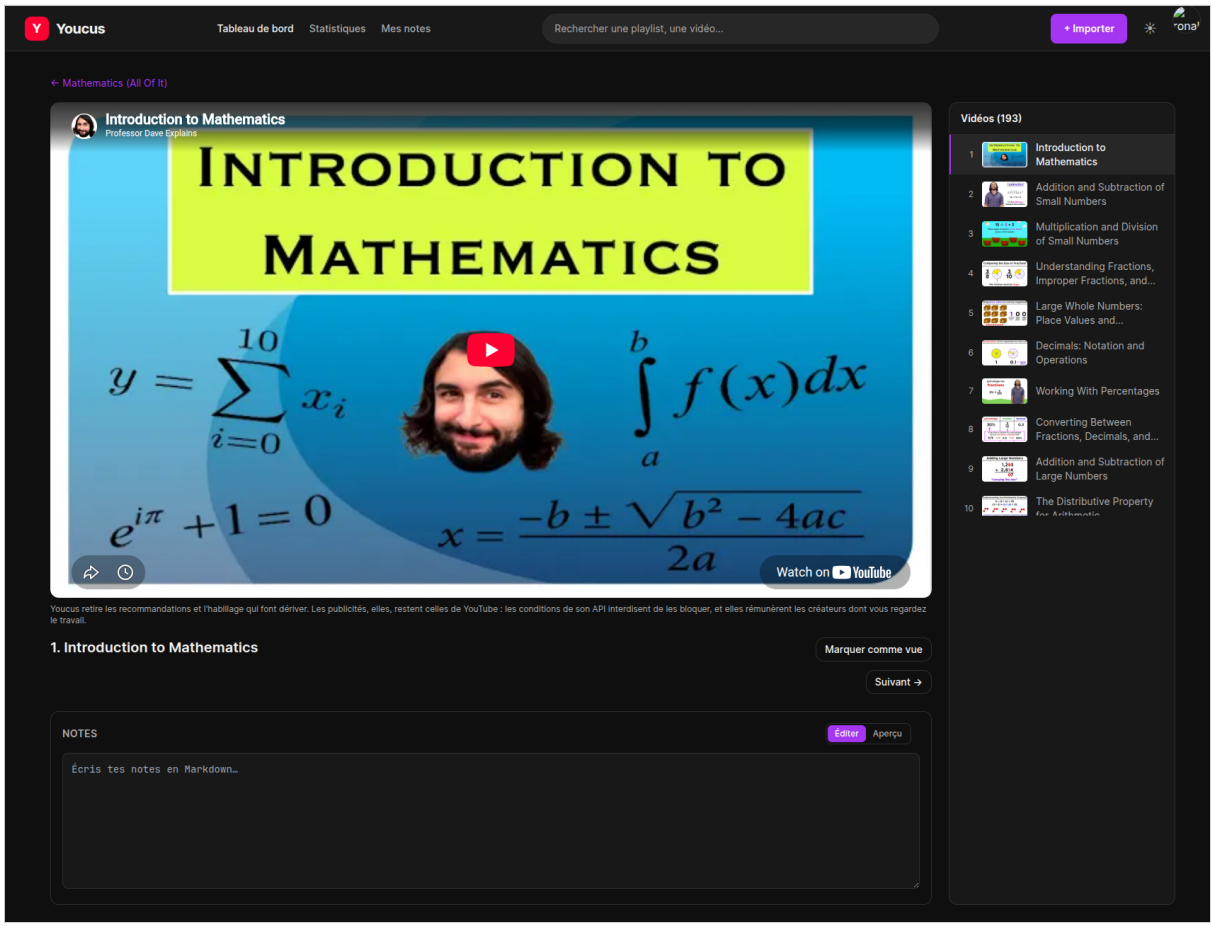


Figure 9. *L'écran final développé : le même lecteur focus en conditions réelles*

9. Conception de la base de données

Le modèle repose sur cinq entités et une table d'association. **User** porte le compte, créé à la première connexion Google. **Playlist** et **Video** représentent les contenus importés depuis YouTube, dont Youcus ne conserve que les métadonnées (titre, miniature, identifiant) conformément aux contraintes de la section 6. **Note** et **Progress** portent ce que l'utilisateur produit : ses notes en Markdown et sa position de lecture. Enfin **PlaylistVideo** relie les playlists aux vidéos.

Cette table d'association est le cœur du modèle. Une même vidéo peut appartenir à plusieurs playlists, et une playlist contient plusieurs vidéos : c'est une relation **N:N**, plusieurs-à-plusieurs, qu'une base relationnelle ne sait pas exprimer directement. On la traduit par une **table de jonction**, avec une ligne par couple playlist-vidéo. C'est aussi le seul endroit où la position d'une vidéo a un sens : la troisième vidéo d'une playlist peut être la dixième d'une autre. La position appartient au lien, pas à la vidéo.

Deux décisions en découlent, que j'assume. La progression et les notes sont **globales par vidéo**, et non par couple playlist-vidéo : une vidéo vue dans une playlist est vue partout, une note suit sa vidéo partout où elle apparaît. C'est cohérent avec l'usage visé : on apprend d'une vidéo, pas d'une vidéo-dans-une-playlist.

J'avais conçu ce modèle ainsi dès le départ, en Merise puis en modèle entités-relations (voir le MCD de conception en annexe). Mais au moment de l'implémentation, dans l'intensité du sprint de développement, j'ai codé sans m'en rendre compte un modèle plus simple : la vidéo rattachée directement à sa playlist en 1:N, position comprise. L'écart ne se voyait pas : l'application fonctionnait, les tests passaient.

Il s'est payé plus tard, sous la forme du bug de cascade décrit à la section 14 : le rafraîchissement d'une playlist détruisait les vidéos pour les recréer, et les suppressions en cascade emportaient les notes et la progression de l'utilisateur. En analysant ce bug, j'ai constaté que sa cause racine n'était pas dans le code du rafraîchissement, mais dans le modèle lui-même. Je suis revenu à ma conception initiale.

Ce retour s'est fait en production, par une migration SQL écrite à la main : celle générée par Prisma supprimait les colonnes sans préserver les données. La mienne procède en six étapes : créer la table de jonction, la remplir en dédupliquant les vidéos par leur identifiant YouTube, fusionner les notes et les progressions des doublons, remapper, nettoyer, poser les contraintes. Les notes en double sont concaténées plutôt qu'écrasées, et la progression retient la valeur la plus avancée. Aucune donnée n'a été perdue.

La leçon que j'en retiens : vérifier l'implémentation contre la conception, pas seulement contre les tests. Les tests validaient ce que le code faisait ; ils ne pouvaient pas voir que le code ne faisait pas ce que la conception avait prévu.

Un point de ce modèle mérite d'être explicité, parce qu'un lecteur attentif le relèvera. La table **Note** porte deux clés étrangères nullables, `videoId` et `playlistId`, et une note doit être rattachée à l'une ou à l'autre, jamais aux deux, jamais à aucune. **Cette contrainte n'est pas exprimée en base : elle est portée par le code.** L'application expose deux chemins distincts, l'un renseignant `videoId`, l'autre `playlistId` ; aucun appel ne peut en renseigner deux. Les index d'unicité garantissent en revanche qu'un auteur n'a qu'une note par cible. Je l'assume comme une limite : une contrainte `CHECK` exigeant qu'exactement une des deux colonnes soit renseignée rendrait la règle vraie indépendamment du code qui l'utilise, et c'est ce que je ferais aujourd'hui. Une invariante de données appartient aux données. Le

MCD de conception initiale figure en annexe A : l'écart entre les deux versions est exactement l'histoire racontée ci-dessus.

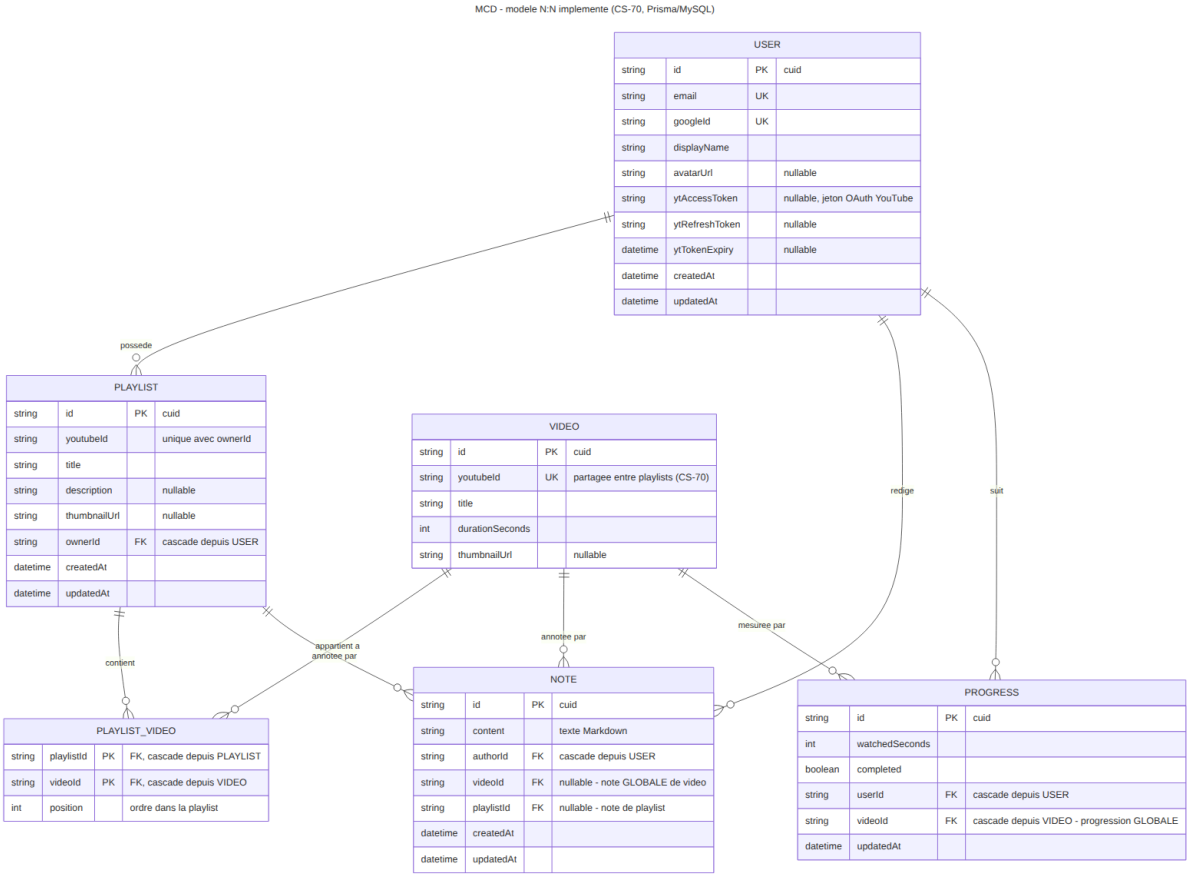


Figure 10. *MCD final : 5 entités, relation N:N Playlist-Video (diagramme 02)*

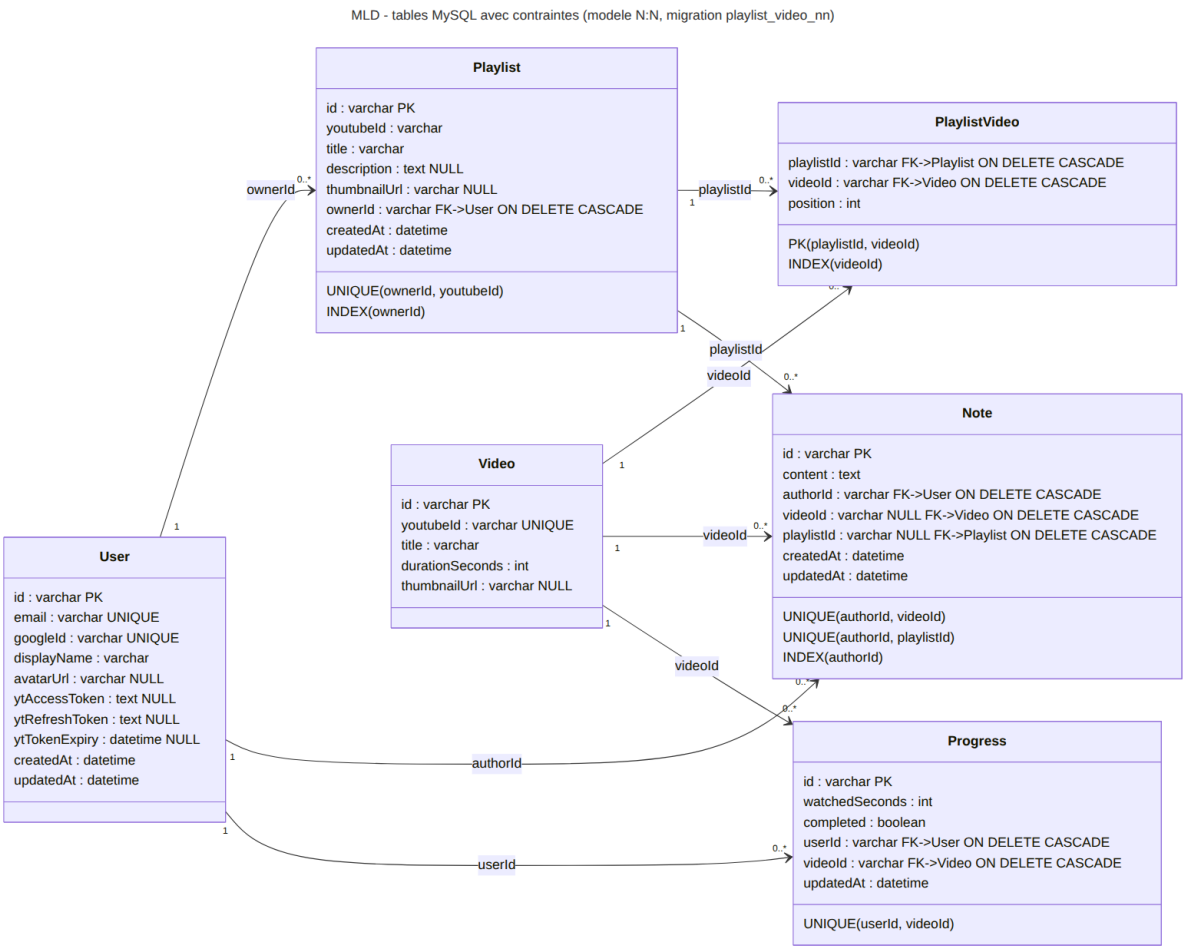


Figure 11. *MLD avec contraintes d'unicité et cascades (diagramme 02b)*

10. Script de création de la base de données

La base est créée et versionnée par les migrations Prisma, appliquées automatiquement au déploiement (`prisma migrate deploy` au démarrage du conteneur). Trois migrations racontent l'histoire du schéma : `0_init` (les 5 tables initiales avec leurs contraintes d'unicité et leurs cascades), `add_youtube_tokens` (l'ajout des jetons OAuth YouTube sur l'utilisateur), et `playlist_video_nn` : la migration du retour au N:N, écrite à la main parce que le SQL généré par l'outil aurait détruit les données : elle crée la jonction, la remplit en dédupliquant les vidéos par youtubeId, remappe les notes et progressions vers les vidéos canoniques (fusion par concaténation pour les notes, par maximum pour les progressions), et seulement ensuite supprime les anciennes colonnes. Le schéma physique est produit et versionné par les migrations Prisma (`server/prisma/migrations/`). Le **script complet de l'état final**, obtenu par `prisma migrate diff --from-empty --to-schema-datamodel prisma/schema --script`, est reproduit en annexe E ; les pages qui suivent en donnent la lecture, table par table, avec la raison de chaque choix. Il correspond au MLD présenté à la section 9, après le refactor de l'association Playlist-Video en N:N.

SGBD : MySQL 8, encodage `utf8mb4` et collation `utf8mb4_unicode_ci` sur toutes les tables : nécessaire pour stocker sans perte les titres de vidéos, qui contiennent couramment des emoji et des alphabets non latins.

Les identifiants sont des `VARCHAR(191)` : ce sont des CUID générés par l'application plutôt que des entiers auto-incrémentés. Ce choix évite d'exposer dans les URL un compteur qui révélerait le volume de données et permettrait d'énumérer les ressources d'autres utilisateurs. La longueur 191 est la limite indexable d'une colonne `utf8mb4` sous l'index `InnoDB` par défaut.

Table `User`

Le compte, créé au premier passage par Google. Aucun mot de passe n'est stocké : l'authentification est déléguée. Les jetons YouTube (accès et rafraîchissement) sont en TEXT car ils dépassent la taille d'un VARCHAR indexable.

Table `Playlist`

Une playlist importée. La contrainte d'unicité porte sur le couple (propriétaire, identifiant YouTube) et non sur l'identifiant seul : deux utilisateurs différents doivent pouvoir importer la même playlist publique.

Table `Video`

Une vidéo YouTube, PARTAGÉE entre toutes les playlists qui la contiennent. C'est le cœur du refactor du 25/07 : `youtubeId` est unique au niveau de la table, plus par playlist.

Table `PlaylistVideo`

La table de jonction de l'association N:N. Sa clé primaire est composite (playlistId, videoId) : une vidéo ne peut apparaître qu'une fois dans une playlist donnée. La position de la vidéo

appartient à cette table, et non à la vidéo : la même vidéo occupe un rang différent selon la playlist.

Table `Note`

Une note Markdown. Elle est rattachée SOIT à une vidéo, SOIT à une playlist : d'où deux clés étrangères nullables et deux contraintes d'unicité distinctes, qui garantissent une seule note par utilisateur et par cible.

Table `Progress`

L'avancement de lecture. L'unicité (userId, videoId) en fait une ligne par utilisateur et par vidéo, mise à jour toutes les cinq secondes pendant la lecture.

Contraintes d'intégrité référentielle

Toutes les clés étrangères sont déclarées `ON DELETE CASCADE`. Ce choix sert la conformité RGPD : la suppression d'un compte efface par construction ses playlists, ses notes et sa progression, sans code applicatif à maintenir en parallèle.

C'est aussi ce comportement qui a causé le bug documenté à la section 14 : tant que le rafraîchissement d'une playlist supprimait puis recréait les lignes `Video`, la cascade détruisait les notes et la progression associées. Le refactor en N:N a supprimé la cause plutôt que le symptôme, le rafraîchissement ne touche plus qu'à la table de jonction, et les vidéos elles-mêmes survivent.

11. Diagramme de cas d’utilisation

L’acteur principal est l’utilisateur authentifié via Google. Ses cas d’utilisation : s’authentifier, gérer son compte (export et suppression RGPD), importer une playlist YouTube (qui inclut la synchronisation), consulter et supprimer ses playlists, les fusionner, lire une vidéo en mode focus (qui inclut la prise de note), suivre sa progression et exporter ses notes.

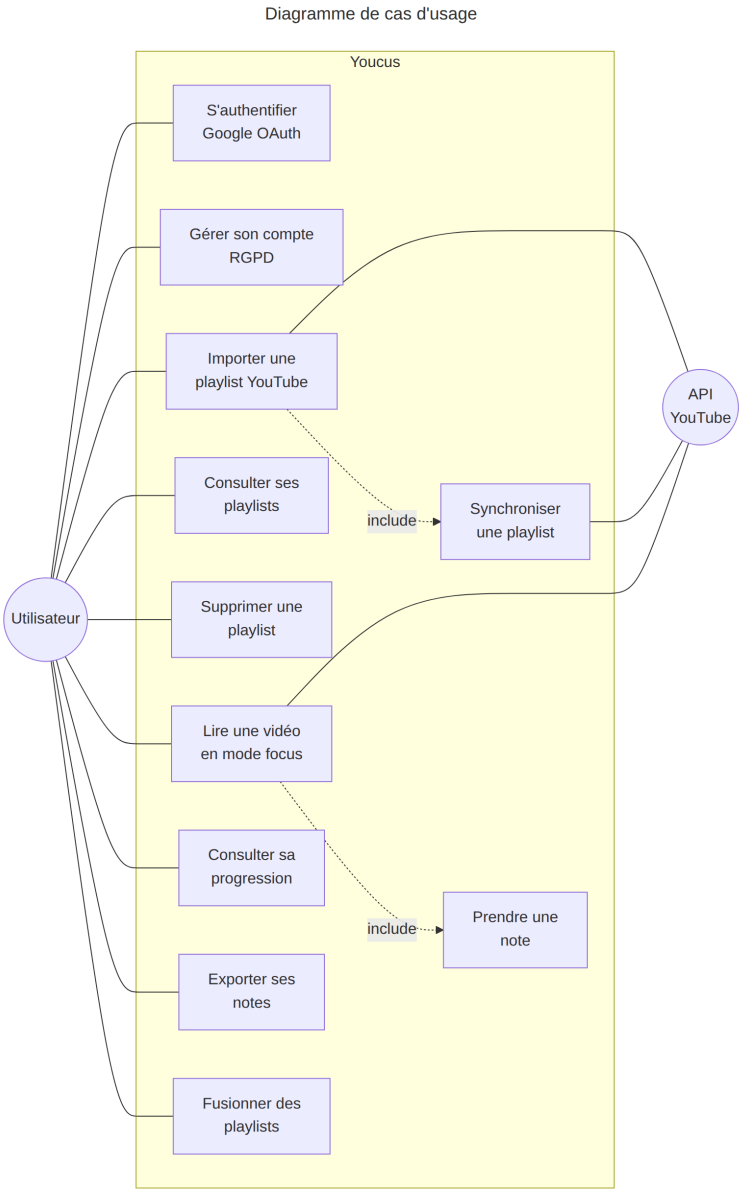


Figure 12. *Diagramme de cas d’utilisation (UML)*

L’API YouTube intervient comme acteur secondaire sur l’import, la synchronisation et la lecture. Le diagramme (01-cas-usage) montre les relations d’inclusion : l’import inclut la synchronisation, la lecture en mode focus inclut la prise de note.

12. Diagrammes de séquence

J'ai retenu les deux flux qui montrent comment l'application se connecte aux systèmes externes : les interactions les plus complexes et les plus risquées du projet, et de fait les sources de mes deux incidents principaux (section 14) : la connexion OAuth Google (cookie d'état anti-CSRF, échange du code contre les jetons, création de session signée) et l'import d'une playlist (pagination de l'API YouTube, transaction d'upsert des vidéos partagées et de la jonction). Les diagrammes 04 et 05 reflètent l'implémentation réelle, vérifiée contre le code.

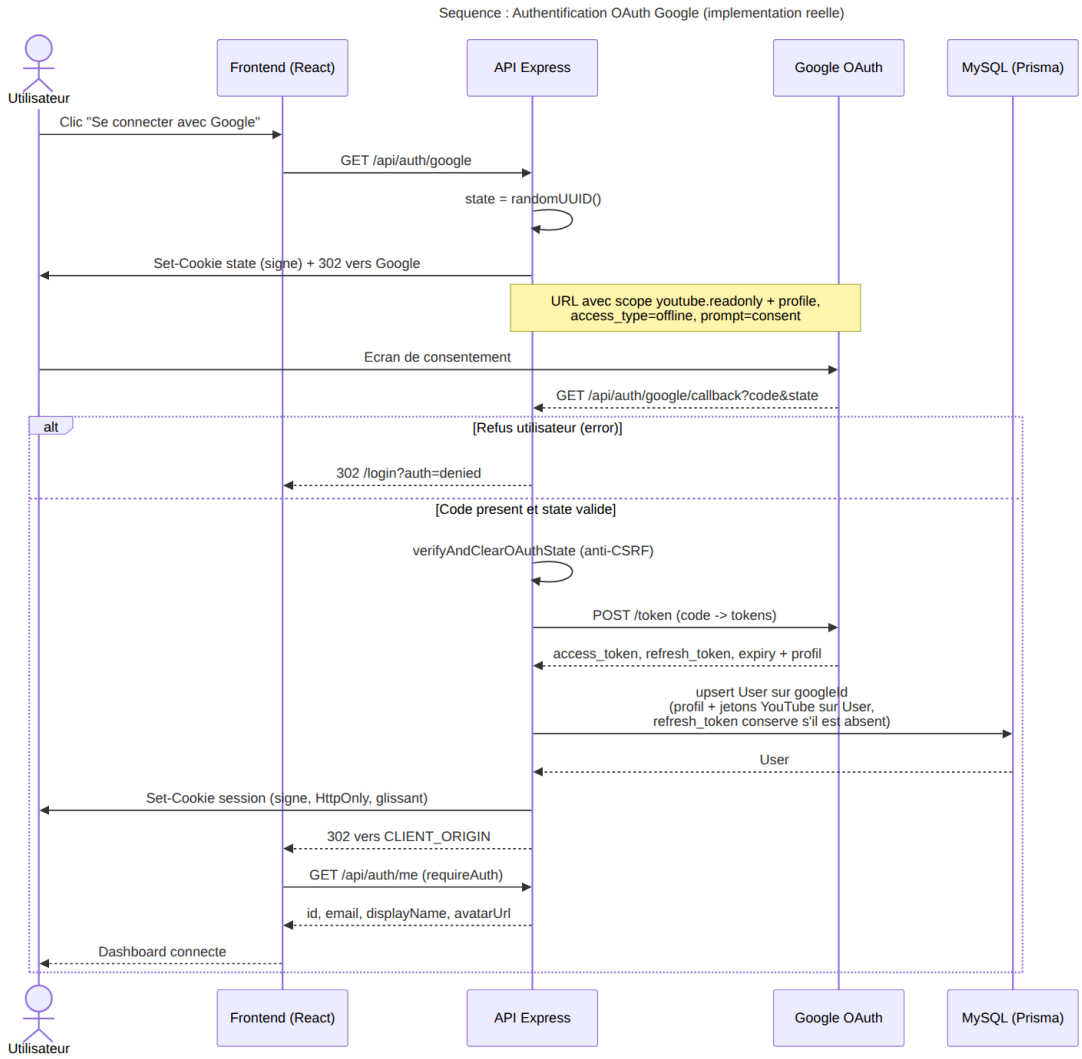


Figure 13. *Diagramme de séquence : connexion OAuth Google (diagramme 04)*

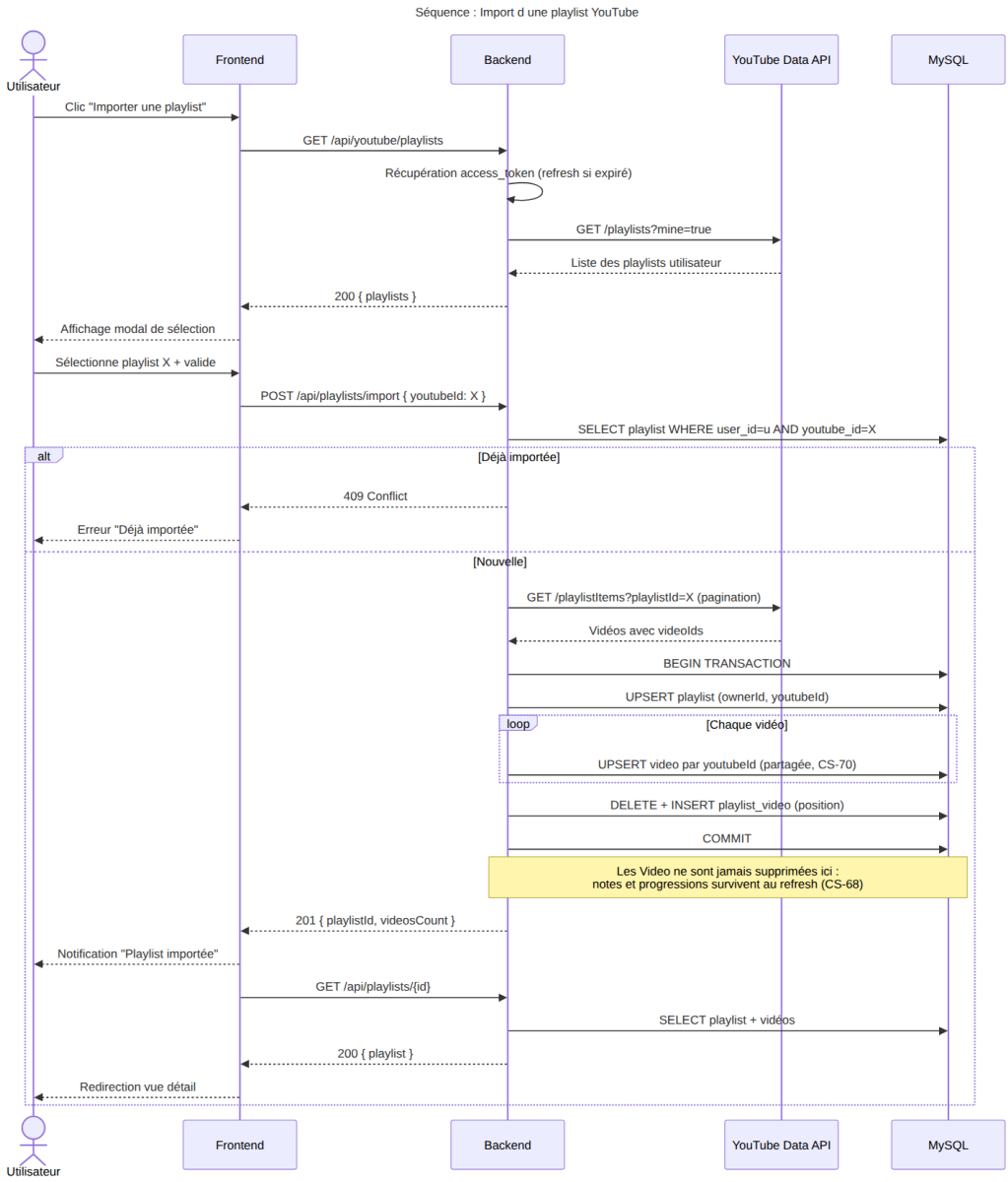


Figure 14. *Diagramme de séquence : import d'une playlist (diagramme 05)*

13. Spécifications techniques

L’ensemble des choix techniques est résumé dans le tableau ci-dessous. Deux choix méritent une phrase : TypeScript de bout en bout pour partager la rigueur de typage entre client et serveur, et le duo MySQL/Redis qui couvre la compétence d’accès aux données relationnelles et NoSQL. Une précision sur la production : la base managée fournie par Sevalla est MariaDB, compatible avec le protocole et le SQL de MySQL. Le développement et l’intégration continue tournent sur MySQL 8, et le schéma Prisma est identique des deux côtés.

DOMAINE	CHOIX
Langages	TypeScript (client et serveur)
Front	React, Vite, TanStack Query, Tailwind CSS
Back	Express, Prisma
Bases de données	MySQL (relationnel), Redis (NoSQL, cache)
Serveurs et déploiement	Docker, Sevalla (API en conteneur, front statique, MariaDB), CI/CD GitHub Actions

14. Réalisation : les difficultés rencontrées

Incident 1 : Le 404 qui accusait la mauvaise coupable

Ce récit réunit ce que j'avais d'abord pris pour deux incidents distincts : les 404 sur l'import et sur `/progress` d'un côté, le hot-reload qui plantait de l'autre. C'était le même problème, et c'est précisément ce qui m'a fait perdre du temps.

Le 19 juillet, l'import d'une playlist échoue. L'interface affiche « Requête échouée », rien ne remonte, et le message ne dit pas pourquoi.

J'ouvre l'onglet Network des outils de développement. La requête `POST` part vers `localhost:5173/api/playlists/import` et revient en **404**. Or `5173` est le port du serveur de développement Vite, celui qui sert le front ; l'API écoute sur `4000`. Ma première lecture est donc naturelle : la requête part au mauvais endroit, il manque une configuration de proxy.

C'était une fausse piste. Le proxy `'/api' → 'http://localhost:4000'` était déjà déclaré dans `vite.config.ts` depuis l'échafaudage du front, la veille au soir. Rien ne manquait.

Le soir même, le même 404 réapparaît à 22h56, cette fois sur `/progress` : une route que je venais de créer une minute plus tôt en commitant CS-18. Deux routes différentes, deux moments différents, un symptôme identique : la cause n'était donc dans aucune des deux.

La vraie explication tenait à un détail du fonctionnement d'un proxy de développement : **il ne peut rediriger que si quelqu'un répond en face.** Quand le serveur d'API est arrêté, la redirection échoue et c'est Vite lui-même qui renvoie un 404. L'URL affichée dans les DevTools : `localhost:5173/...` : ne signifiait pas que ma requête partait au mauvais endroit, mais que **personne n'était là pour la recevoir**.

Et mon serveur était bel et bien planté. Le hot-reload de développement (`nodemon` + `tsx`) crashait sur `ERR_MODULE_NOT_FOUND: src/index.ts` en affichant « app crashed », sans que je le remarque : je regardais le navigateur, pas le terminal du serveur. Chaque fois que je modifiais le back, le processus mourait, et la requête suivante depuis le front revenait en 404.

J'ai traité la cause plutôt que le symptôme : plutôt que de relancer le serveur à la main à chaque fois, j'ai ouvert le ticket **CS-56 « Fiabiliser le hot-reload du serveur de dev »**, corrigé la configuration de `nodemon`, et fermé le ticket le soir même à 23h14. Les 404 ont disparu avec.

Ce que j'en retiens, et qui a changé ma façon de déboguer : un `404` ne dit pas « cette route n'existe pas », il dit « rien n'a répondu à cette adresse ». Derrière un proxy de développement, les deux situations produisent exactement la même erreur. Le réflexe que j'ai acquis est de vérifier que le serveur tourne **avant** de suspecter le code, et de garder le terminal du back visible pendant que je développe le front, plutôt que de découvrir son crash par ricochet, une heure plus tard.

C'est aussi un exemple de ma façon de travailler seul : une gêne d'outillage répétée n'est pas une fatalité qu'on contourne, elle devient un ticket dans le backlog, traité comme le reste.

Preuves : `c14b-bug-import-404-ecran.png`, `c14b-bug-import-404-devtools-headers.png` (le POST vers 5173), `c14b-bug-import-404-logs-pino.png`, `c14d-bug-progress-404.png` (la récidive à 22h56), `c14e-hot-reload-crash-cs56.png` (« app

crashed » : la vraie cause). Chronologie git : CS-18 commité à 22h55, 404 à 22h56, CS-56 fermé à 23h14.

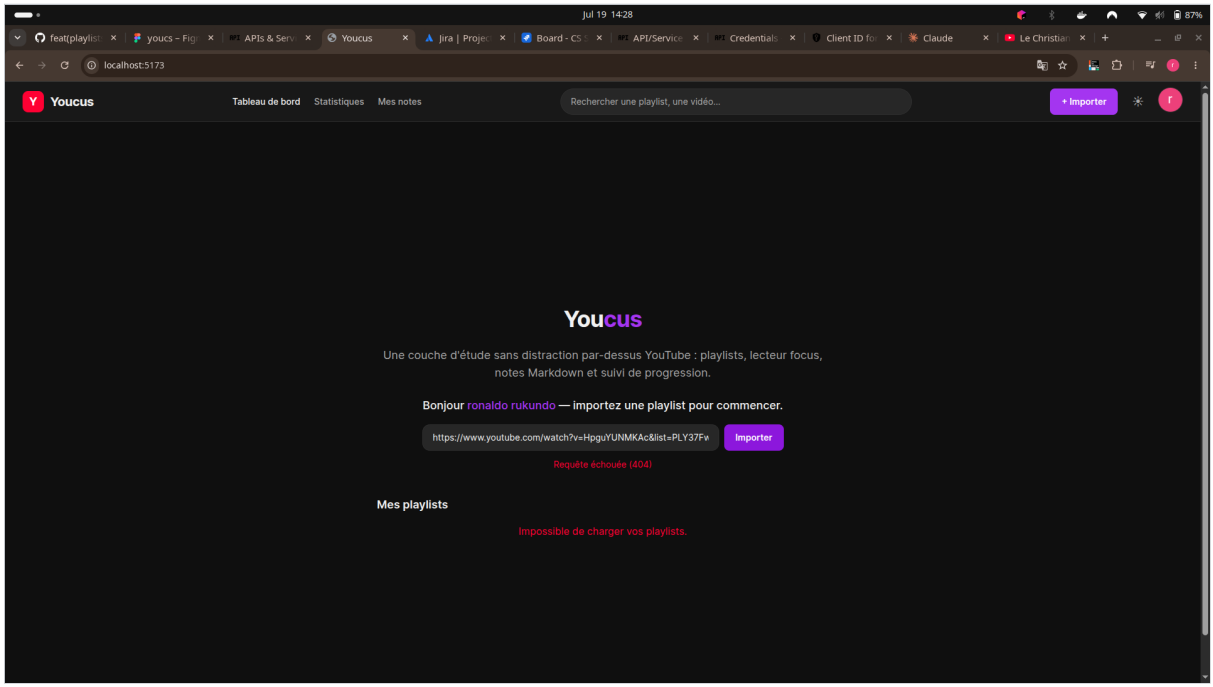


Figure 15. *c14b bug import 404 ecran*

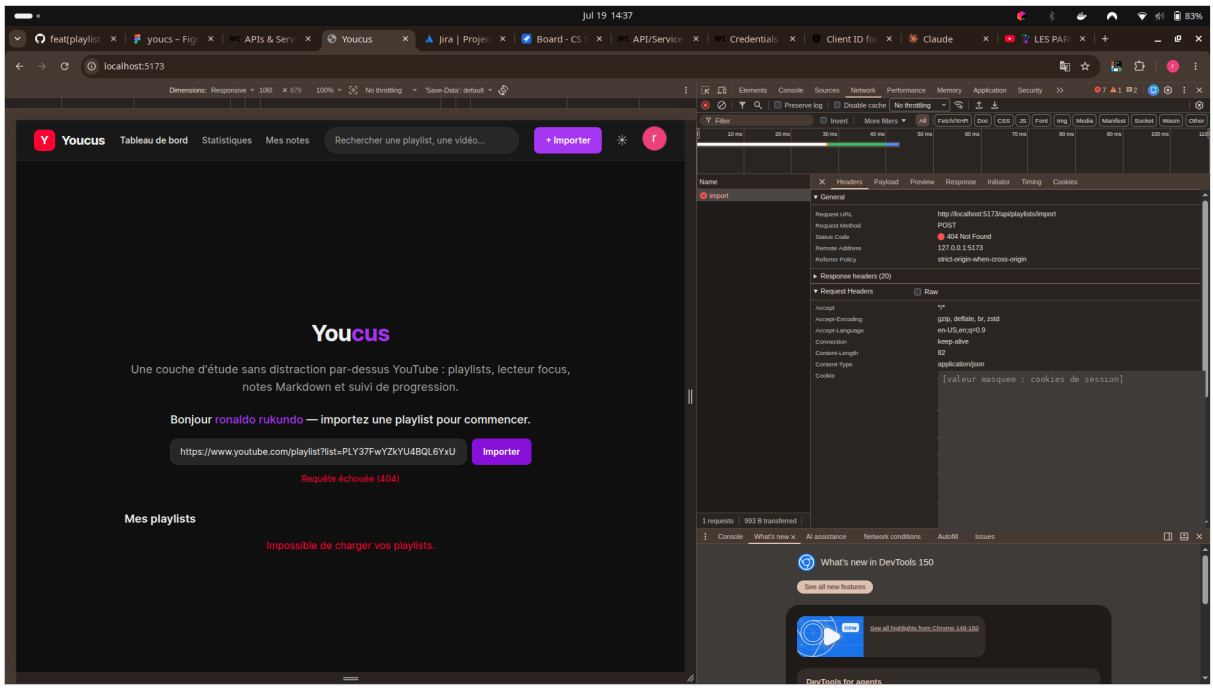


Figure 16. *c14b bug import 404 devtools headers*

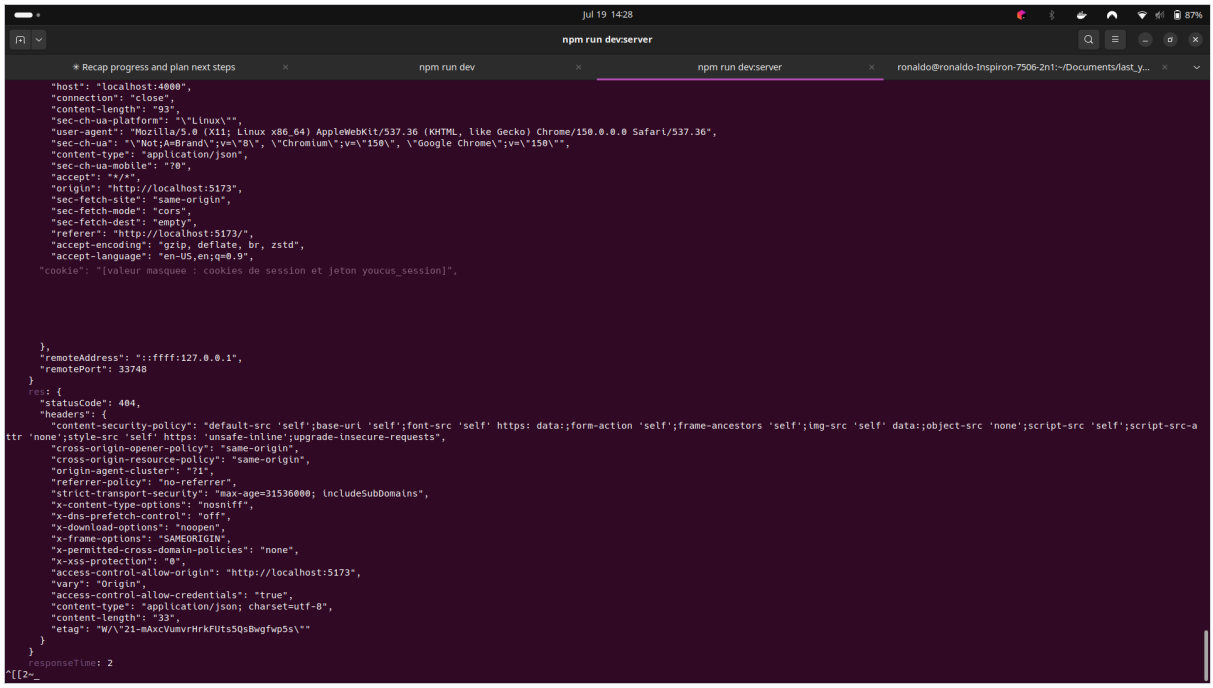


Figure 17. *c14b bug import 404 logs pino*

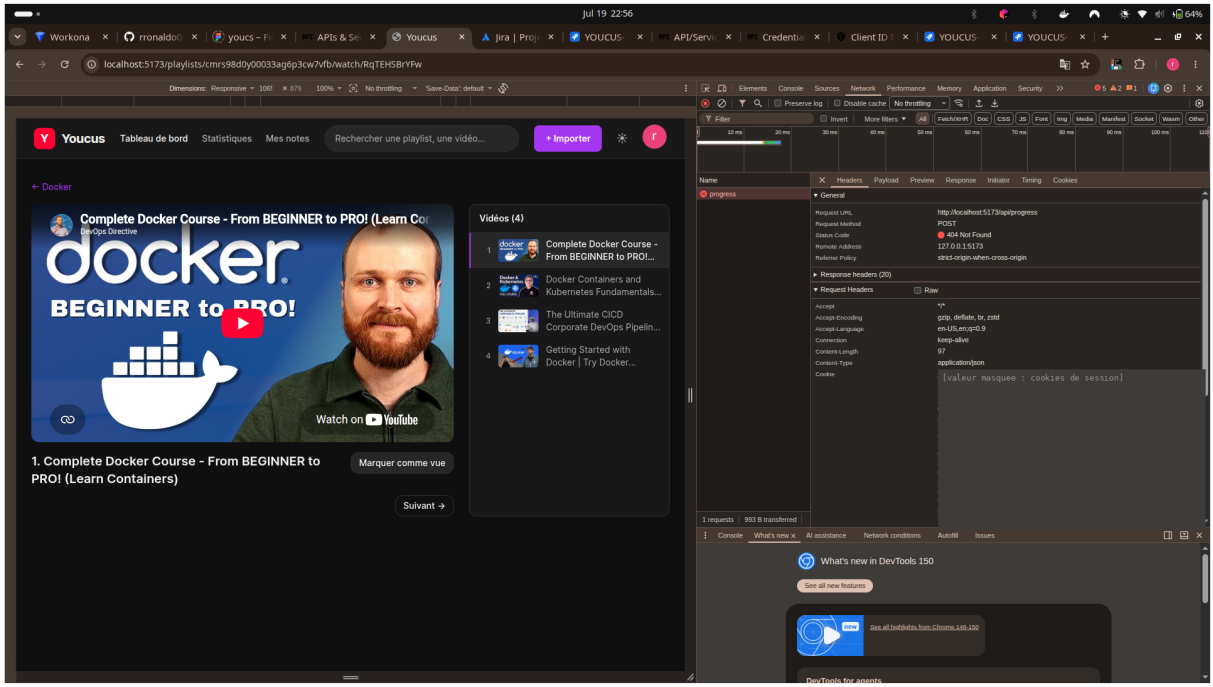


Figure 18. *c14d bug progress 404*

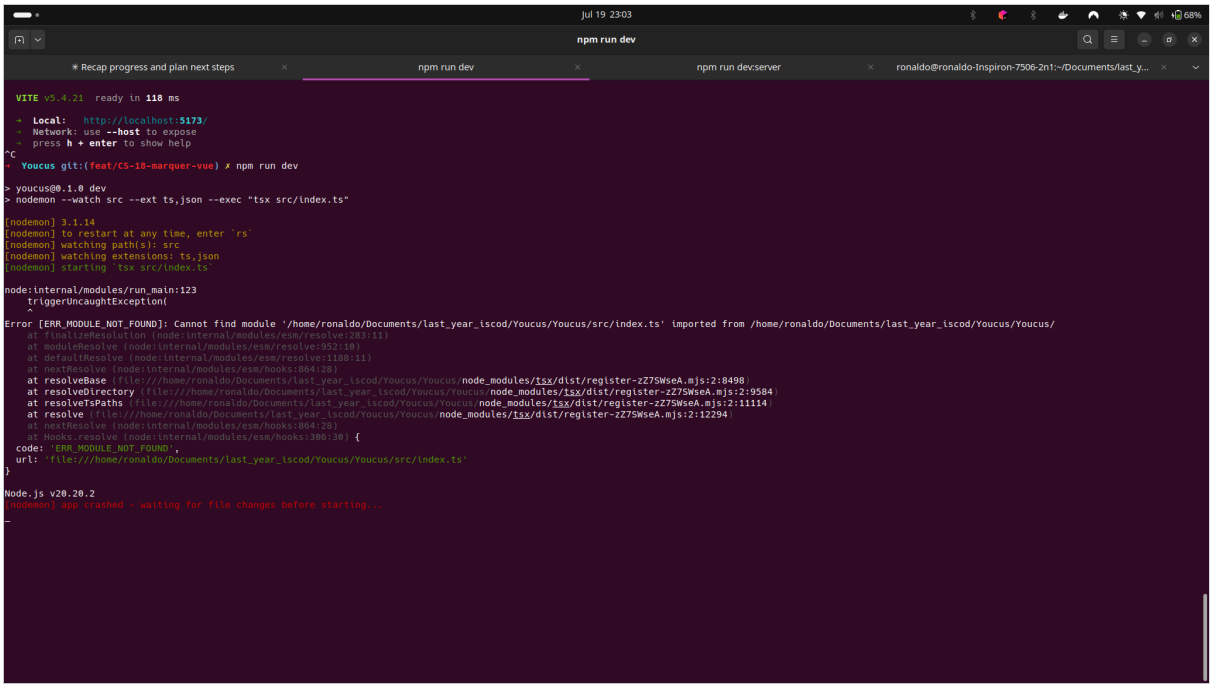


Figure 19. *c14e hot reload crash cs56*

Incident 2 : Les deux adresses qu'on confond

L'authentification Google repose sur un aller-retour : mon application envoie l'utilisateur chez Google, et Google le renvoie ensuite vers une adresse que j'ai déclarée à l'avance, la *redirect URI*. Si l'adresse que mon serveur annonce ne correspond pas exactement à celle enregistrée dans la console Google Cloud, Google refuse la connexion. C'est une sécurité : elle empêche un tiers de détourner le retour d'authentification vers un site qu'il contrôle.

Mon erreur a été d'inverser deux adresses qui se ressemblent beaucoup dans ma configuration :

```
CLIENT_ORIGIN        = http://localhost:5173                          (le
front)
GOOGLE_CALLBACK_URL  = http://localhost:4000/api/auth/google/callback
(l'API)
```

J'avais mis l'une à la place de l'autre. L'erreur est facile à faire parce qu'elle heurte l'intuition : on pense « l'utilisateur revient sur mon site », donc on écrit l'adresse du site. Mais ce n'est pas l'utilisateur qui revient, c'est **Google qui appelle mon API** avec un code d'autorisation à échanger contre un jeton. Le retour doit donc pointer vers le serveur, pas vers l'interface : vers le port 4000, pas le 5173.

Le même piège s'est reposé au déploiement : l'adresse de production n'est pas celle de développement. Une redirect URI déclarée pour `localhost` ne vaut rien une fois l'application sur Sevalla. J'ai donc déclaré **les deux** dans la console Google Cloud, `localhost:4000` et le domaine de production, et rendu l'adresse configurable par variable d'environnement (`GOOGLE_CALLBACK_URL`) plutôt que codée en dur : c'est ce que montre la capture de mes identifiants OAuth, et c'est la trace de la leçon.

Ce que j'en retiens : quand deux valeurs de configuration se ressemblent et désignent des choses différentes, l'erreur n'est pas une étourderie, elle est prévisible. La parade n'est pas d'être plus attentif, c'est de rendre la confusion impossible : nommer explicitement

(`CLIENT_ORIGIN` contre `GOOGLE_CALLBACK_URL`), commenter le fichier d'exemple, et faire valider la configuration au démarrage plutôt que de découvrir l'erreur au premier clic d'un utilisateur.

Suite de l'incident, le 26/07 : en préparant ce dossier, je me suis aperçu qu'une capture d'écran destinée à l'illustrer laissait apparaître mon `GOOGLE_CLIENT_SECRET` en clair. J'ai fait tourner le secret dans la console Google Cloud, puis je suis allé plus loin que la simple correction : j'ai créé **une seconde application Google dédiée à la production**, de sorte que les identifiants de développement et de production soient désormais entièrement séparés. Une fuite sur l'un ne compromet plus l'autre.

Preuves : `c14a-google-oauth-client-redirect-uris.png` (les deux redirect URIs déclarées), `c14a-google-cloud-api-key.png` (création de la clé YouTube). Code : `server/src/lib/googleOAuth.ts` (l'URL lue depuis l'environnement), `server/src/config/env.ts` (validation de la configuration au démarrage).

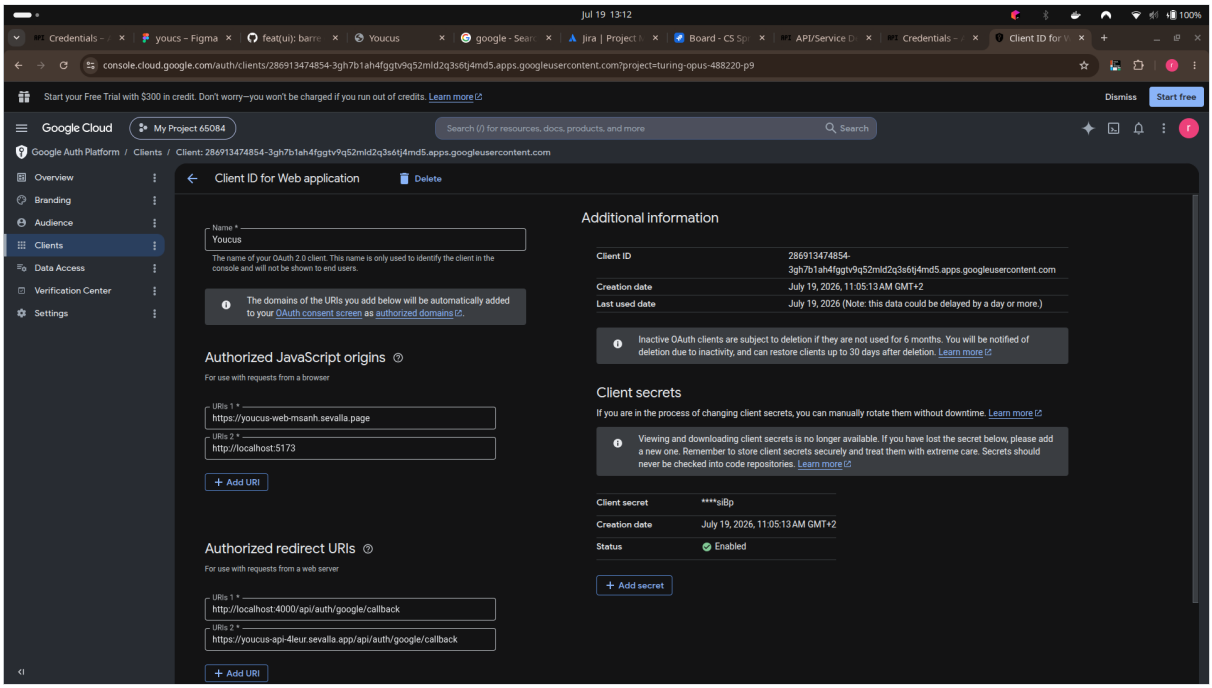


Figure 20. *c14a google oauth client redirect uris*

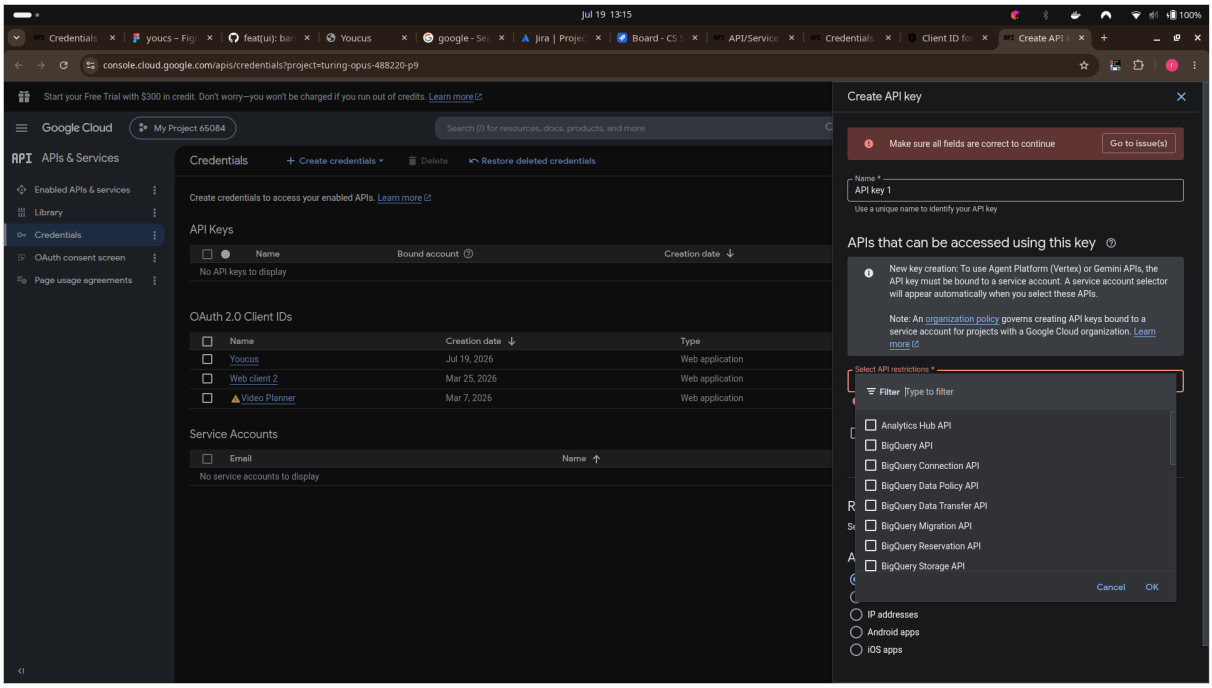


Figure 21. *c14a google cloud api key*

Incident 3 : La dépendance oubliée, et ce qu'elle m'a fait ajouter

Le 20 juillet, vers 16 h, en développant l'aperçu Markdown des notes (CS-17), Vite affiche un bandeau rouge en plein écran :

```
Failed to resolve import "react-markdown" from "src/features/notes/
VideoNotes.tsx"
```

J'avais écrit l'`import` avant de lancer le `npm install`. La correction a pris dix secondes. Si je fais figurer cet incident dans ce dossier malgré sa banalité, c'est pour deux raisons.

La première : **le message d'erreur a tout dit**. Le nom du module, le fichier fautif, la nature du problème. Aucune enquête n'était nécessaire, contrairement au 404 de l'incident 1 où le message accusait la mauvaise coupable. Savoir reconnaître lequel des deux cas on a sous les yeux : un message qui dit la vérité, ou un message qui dit ce que la couche intermédiaire a cru comprendre : est ce qui décide si on corrige en dix secondes ou si on part enquêter.

La seconde : **c'est en installant la dépendance que j'ai posé la question de sécurité**. Afficher du Markdown écrit par l'utilisateur, c'est accepter d'injecter du HTML généré dans la page : la porte d'entrée classique d'une faille XSS. J'ai donc installé `rehype-sanitize` en même temps, et le rendu passe par un filtre qui retire tout ce qui pourrait s'exécuter :

```
<ReactMarkdown rehypePlugins={[rehypeSanitize]}>{draft}</ReactMarkdown>
```

L'incident n'a donc rien coûté, mais il a servi de déclencheur : le réflexe utile n'est pas « installer ce qui manque », c'est **se demander ce que la bibliothèque va faire de données que je ne contrôle pas**.

Preuve : `c14f-bug-react-markdown-manquant.png`. Code : `client/src/features/notes/NoteEditor.tsx` (lignes 2-3 et 120), `client/package.json`.

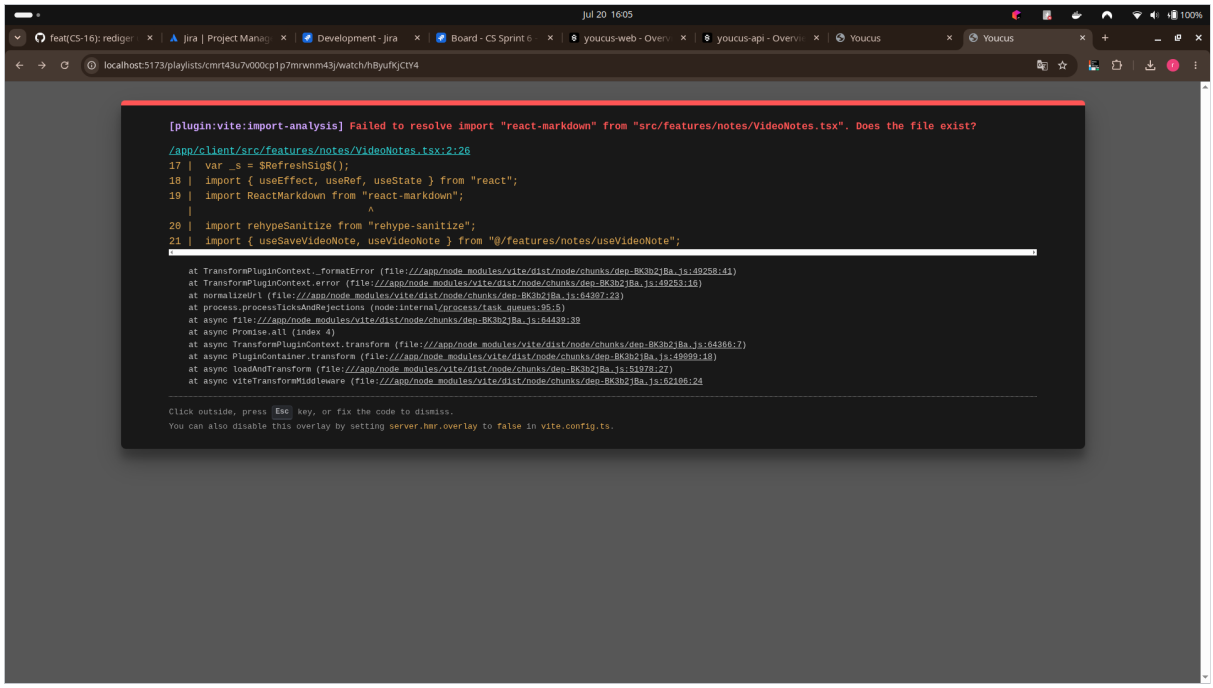


Figure 22. *c14f bug react markdown manquant*

Le lecteur focus : une contrainte produit avant une contrainte technique

Youcus ne cherche pas à remplacer YouTube ni à priver l'utilisateur de son compte. L'objectif est plus modeste et plus précis : **lui offrir un endroit où regarder ce qu'il a choisi de regarder**, sans la colonne de suggestions qui transforme une session d'apprentissage de vingt minutes en deux heures de dérive. Il garde ses playlists chez YouTube, son historique, son algorithme ; il vient simplement les consulter dans un cadre qui ne l'attire pas ailleurs.

Techniquement, cela passe par l'**IFrame Player API** de YouTube, chargée dynamiquement, et par une configuration du lecteur qui retire ce qui distrait :

```
playerVars: {
  rel: 0,                  // suggestions limitées à la même chaîne
  modestbranding: 1,       // habillage YouTube réduit
  iv_load_policy: 3,       // annotations désactivées
}
```

Une limite que j'assume et que je documente : `rel: 0` ne supprime plus les vidéos suggérées depuis un changement d'API de 2018 : il les restreint aux vidéos de la même chaîne. Et si l'utilisateur est connecté à son compte Google dans son navigateur, la lecture dans l'iframe est comptabilisée par YouTube comme n'importe quelle autre. Ce que Youcus protège réellement, c'est donc **l'attention**, pas les données. Le distinguer explicitement m'a paru plus solide que de promettre une étanchéité que l'intégration d'un lecteur tiers ne permet pas.

La reprise de lecture (CS-19) illustre l'autre versant du même choix. Comme le lecteur est une iframe hébergée par YouTube, mon application ne peut pas lire la position de lecture à volonté : elle doit s'abonner aux événements du lecteur. Le composant remonte donc la position toutes les cinq secondes, mais **seulement pendant la lecture effective** :

```
onStateChange: (e) => {
  if (e.data === PLAYING) interval = setInterval(report, 5000)
  else clearInterval(interval)
}
```

Cinq secondes est un compromis assumé : plus court, on inonde l'API d'écritures pour un gain imperceptible ; plus long, l'utilisateur qui ferme brutalement son onglet perd jusqu'à une demi-minute de progression. Et arrêter l'intervalle dès que la lecture s'interrompt évite d'écrire en boucle la même valeur pendant qu'une vidéo est en pause : un détail, mais c'est exactement le genre de détail qui fait la différence entre une fonctionnalité qui marche et une fonctionnalité qui tient en production.

Une contrainte contractuelle, pas seulement technique. Les publicités continuent de s'afficher dans le lecteur intégré : leur diffusion dépend de la monétisation choisie par l'auteur de la vidéo et des systèmes publicitaires de Google, et aucun paramètre de l'IFrame Player API ne permet de les désactiver. Surtout, les *YouTube API Services Terms of Service* interdisent explicitement de restreindre, bloquer, modifier ou remplacer les publicités diffusées par le lecteur : un service qui le ferait perdrait son accès à l'API. J'ai donc renoncé à toute tentative en ce sens, et j'ai resserré la promesse du produit en conséquence : **Youcus supprime la dérive, pas la publicité**. Un utilisateur qui souhaite une lecture sans publicité relève de YouTube Premium, pas de mon application.

La décision produit qui en découle (CS-71) est de **l'afficher plutôt que de le taire** : une mention discrète sous le lecteur explique que les recommandations et l'habillage ont été retirés, que les publicités restent celles de YouTube, et pourquoi, elles rémunèrent les créateurs dont l'utilisateur regarde le travail.

Ce n'est pas une excuse : une plateforme qui vit de contenus gratuits produits par d'autres n'a pas de raison légitime de couper leur revenu. Dire à l'utilisateur ce que l'application fait et ne fait pas vaut mieux qu'une promesse implicite qu'il découvrirait fausse à la première publicité.

Ce point mérite d'être souligné parce qu'il illustre une compétence qui n'est pas technique : lire les conditions d'utilisation d'un service tiers avant de bâtir une fonctionnalité dessus, et accepter qu'une contrainte contractuelle redéfinisse le périmètre du produit.

Code : `client/src/features/player/FocusPlayer.tsx`, `client/src/features/player/VideoSidebar.tsx` (navigation entre les vidéos d'une playlist, CS-15). Tests : `FocusPlayer.test.tsx`, `VideoSidebar.test.tsx`.

15. Interfaces et composants de présentation

Captures : `c15a-lecteur-focus.png` et `c15b-editeur-notes-apercu.png` (relevées le 26/07/2026 sur la stack de développement, avec une playlist réelle de 193 vidéos).

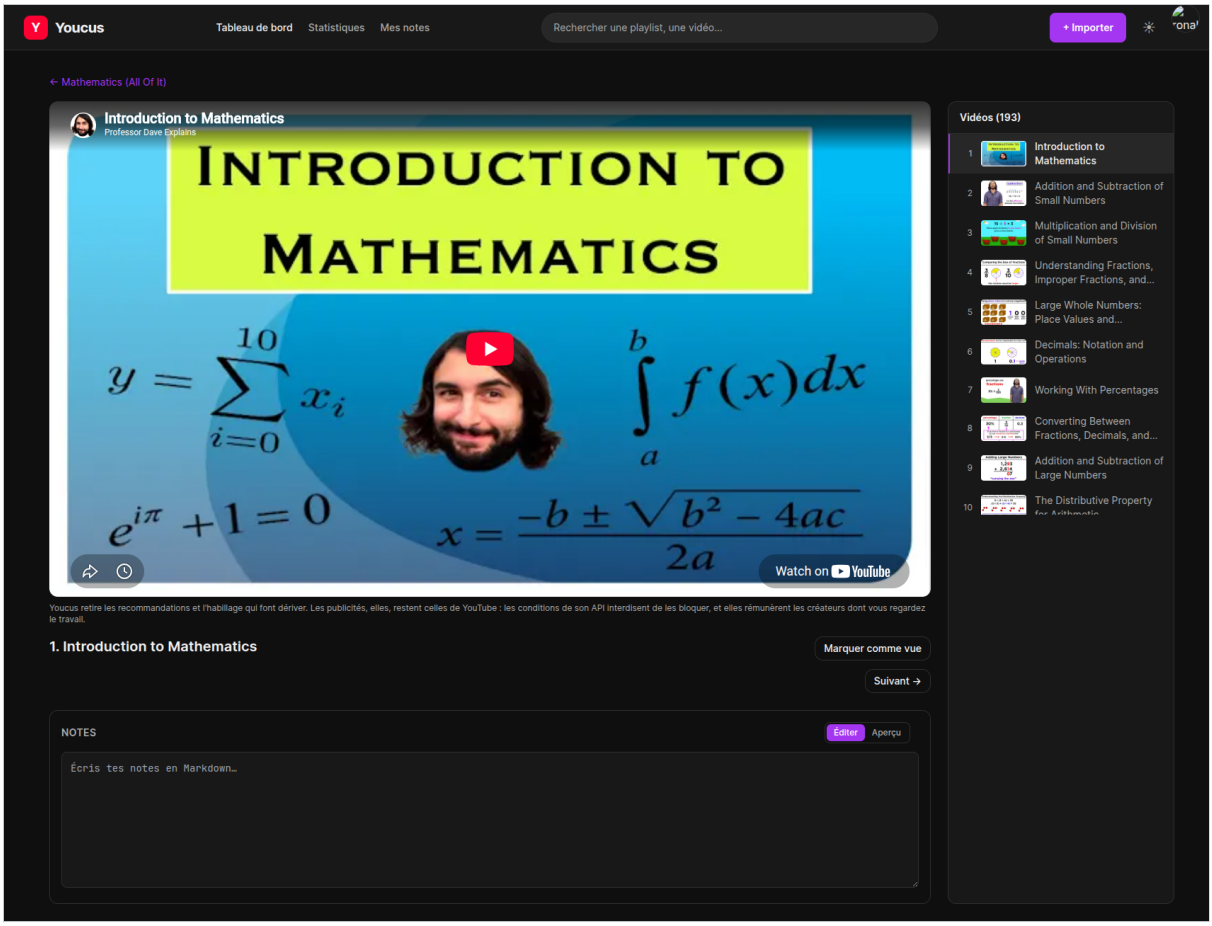


Figure 23. *Le lecteur focus : playlist réelle de 193 vidéos*

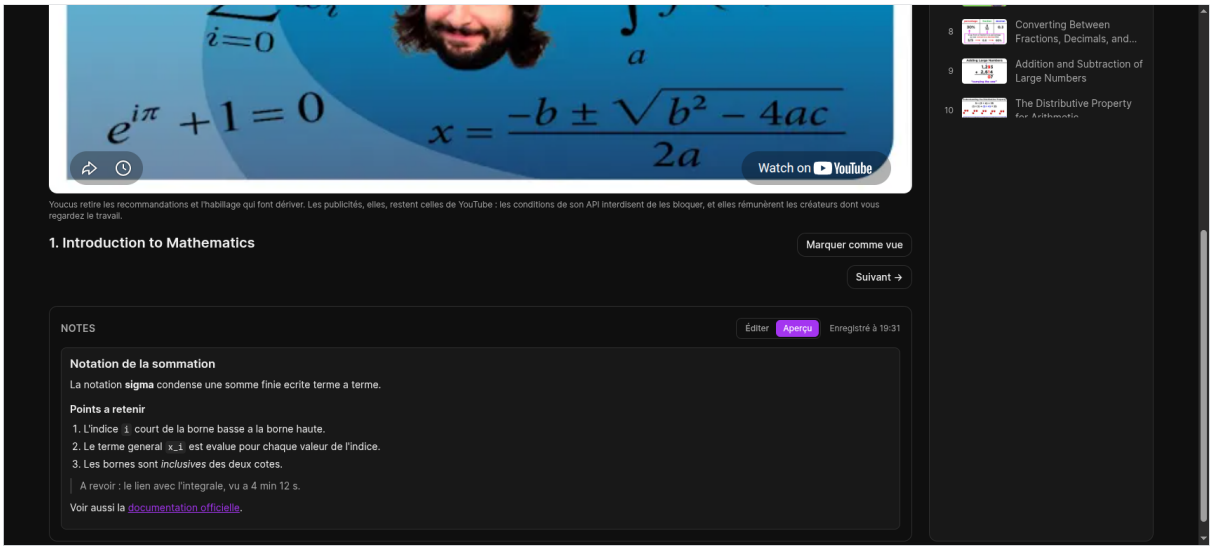


Figure 24. *L'éditeur de notes Markdown et son aperçu assaini*

La première montre le **lecteur focus** : la vidéo en cours, la liste latérale des 193 vidéos de la playlist avec la position courante mise en évidence, les commandes « Marquer comme

vue » et « Suivant », et sous le lecteur la mention de transparence sur les publicités (CS-71). Ce qui n'y figure pas est aussi important que ce qui s'y trouve : ni colonne de suggestions, ni fil de commentaires, ni bandeau de chaîne.

La seconde montre l'**éditeur de notes en mode Aperçu**, où le Markdown saisi par l'utilisateur est rendu : titres, gras, italique, code en ligne, liste ordonnée, citation, lien. La mention « Enregistré à 19:31 » à côté du sélecteur Éditer/Aperçu est la trace de la sauvegarde automatique. C'est ce rendu qui est assaini par `rehype-sanitize` (voir section 19).

Le composant `FocusPlayer` illustre la manière dont l'interface consomme une API tierce sans lui abandonner le contrôle. Le lecteur YouTube vit dans une iframe que je ne peux pas inspecter : je ne peux ni lire la position à volonté, ni savoir quand l'utilisateur met en pause. Le composant s'abonne donc aux événements du lecteur et n'écrit que pendant la lecture effective.

```
onStateChange: (e: { data: number }) => {
  if (e.data === window.YT.PlayerState.PLAYING) {
    interval = setInterval(report, 5000)   // on remonte la position toutes
les 5 s
  } else {
    clearInterval(interval)                // pause, fin, buffering : on
arrête d'écrire
  }
}
```

Le nettoyage est fait dans le retour du `useEffect` : l'intervalle est annulé et le lecteur détruit quand le composant disparaît. Sans cela, changer de vidéo laisserait derrière soi un intervalle qui écrirait la progression d'une vidéo qu'on ne regarde plus, et un lecteur fantôme en mémoire.

```
return () => {
  cancelled = true
  if (interval) clearInterval(interval)
  player?.destroy()
}
```

Le drapeau `cancelled` répond à un problème de course propre à React : le chargement de l'API YouTube est asynchrone, et le composant peut être démonté avant sa résolution. Sans ce drapeau, la promesse construirait un lecteur dans un conteneur qui n'existe plus.

Accessibilité : ce qui est fait, ce qui ne l'est pas

L'interface repose sur du HTML sémantique et sur les composants natifs du navigateur : les actions sont de vrais boutons et les liens de vrais liens, ce qui leur donne gratuitement le focus clavier, l'activation au clavier et une restitution correcte par un lecteur d'écran. Les champs sont associés à leur étiquette, les images portent un texte alternatif, et la navigation principale reste atteignable au clavier seul.

Je m'arrête là où s'arrête ce que j'ai vérifié. **Je n'ai pas conduit d'audit d'accessibilité**, ni mesuré les rapports de contraste, ni testé le parcours au lecteur d'écran, ni vérifié la conformité au RGAA ou aux critères WCAG : l'annoncer serait une affirmation invérifiable, et ce dossier s'interdit ce genre d'affirmation. Le correctif est identifié : une analyse automatique des contrastes et des attributs ARIA manquants, puis une reprise à la main du parcours d'import et de prise de notes, les deux fonctions qui font l'intérêt du produit.

16. Composants métier

La synchronisation d'une playlist : l'endroit où le bug a été tué

`syncVideos` est le cœur métier de l'application et la fonction qui a fait l'objet du refactor le plus important du projet. Sa version d'origine supprimait puis recréait les lignes `Video` ; les cascades du schéma détruisaient alors les notes et les progressions de l'utilisateur à chaque rafraîchissement.

La version actuelle ne touche plus jamais aux vidéos elles-mêmes. Elle les **apparie** par leur identifiant YouTube et ne réécrit que la table de jonction :

```
for (const v of unique.values()) {
  const video = await tx.video.upsert({
    where:  { youtubeId: v.youtubeId },                                    //
appariement
    create: { youtubeId: v.youtubeId, title: v.title, thumbnailUrl:
v.thumbnailUrl },
    update: { title: v.title, thumbnailUrl: v.thumbnailUrl },              //
titre rafraîchi
  })
  rows.push({ playlistId, videoId: video.id, position: v.position })
}

await tx.playlistVideo.deleteMany({ where: { playlistId } })   // seule la
jonction est réécrite
if (rows.length > 0) await tx.playlistVideo.createMany({ data: rows })
```

Trois décisions y sont visibles :

1. **`upsert` plutôt que `create`** : une vidéo présente dans deux playlists n'existe qu'une fois en base, et son titre est mis à jour si l'auteur l'a renommée sur YouTube.
2. **La déduplication en amont** par une `Map` : une playlist YouTube peut contenir deux fois la même vidéo, ce que la clé primaire composite de la jonction interdit. Je garde la première occurrence plutôt que de laisser la base lever une erreur.
3. **Tout se passe dans une transaction** (`tx`), passée en paramètre par l'appelant. Un rafraîchissement interrompu à mi-course ne laisse pas une playlist à moitié vidée.

Le tout est appelé dans `refreshPlaylist`, qui commence par le contrôle d'accès et refuse explicitement de rafraîchir une playlist fusionnée : celle-ci n'a pas d'équivalent chez YouTube :

```
const existing = await prisma.playlist.findFirst({ where: { id, ownerId:
userId } })
if (!existing) throw new HttpError(404, 'Playlist introuvable')
if (existing.youtubeId.startsWith('merge:')) {
  throw new HttpError(400, 'Une playlist fusionnée ne peut pas être
rafraîchie')
}
```

Le `404` sur une playlist appartenant à quelqu'un d'autre est délibéré : répondre `403` confirmerait à un attaquant que l'identifiant existe.

L'export RGPD

`exportUserData` rassemble en une requête toutes les données personnelles, en traversant les relations dans le bon ordre pour restituer les vidéos à leur position réelle dans chaque playlist :

```
include: {
  playlists: { include: { videos: { orderBy: { position: 'asc' }, include:
{ video: true } } } },
  progress: true,
  notes: true,
}
```

L’export **exclut volontairement les jetons OAuth YouTube** et ne renvoie qu’un booléen `youtubeConnected`. Le droit d’accès porte sur les données personnelles, pas sur les secrets d’authentification : les inclure transformerait un fichier téléchargeable en vecteur de compromission du compte Google de l’utilisateur.

17. Composants d’accès aux données

L’accès relationnel par Prisma

L’accès aux données passe par **Prisma**, qui joue le rôle de couche d’accès : le schéma déclaré en `.prisma` génère un client typé, et chaque service consomme ce client sans jamais écrire de SQL. Les bénéfices sont concrets : requêtes paramétrées (donc pas d’injection SQL), types dérivés du schéma (une colonne renommée casse la compilation plutôt que la production), et transactions explicites.

Le contrôle d’accès est intégré à la couche d’accès plutôt qu’ajouté au-dessus, y compris pour les ressources atteintes indirectement :

```
// une note n'est lisible que si sa vidéo appartient à une playlist de
l'utilisateur
where: { id: videoId, playlists: { some: { playlist: { ownerId:
userId } } } }
```

Le cache Redis : la couche NoSQL

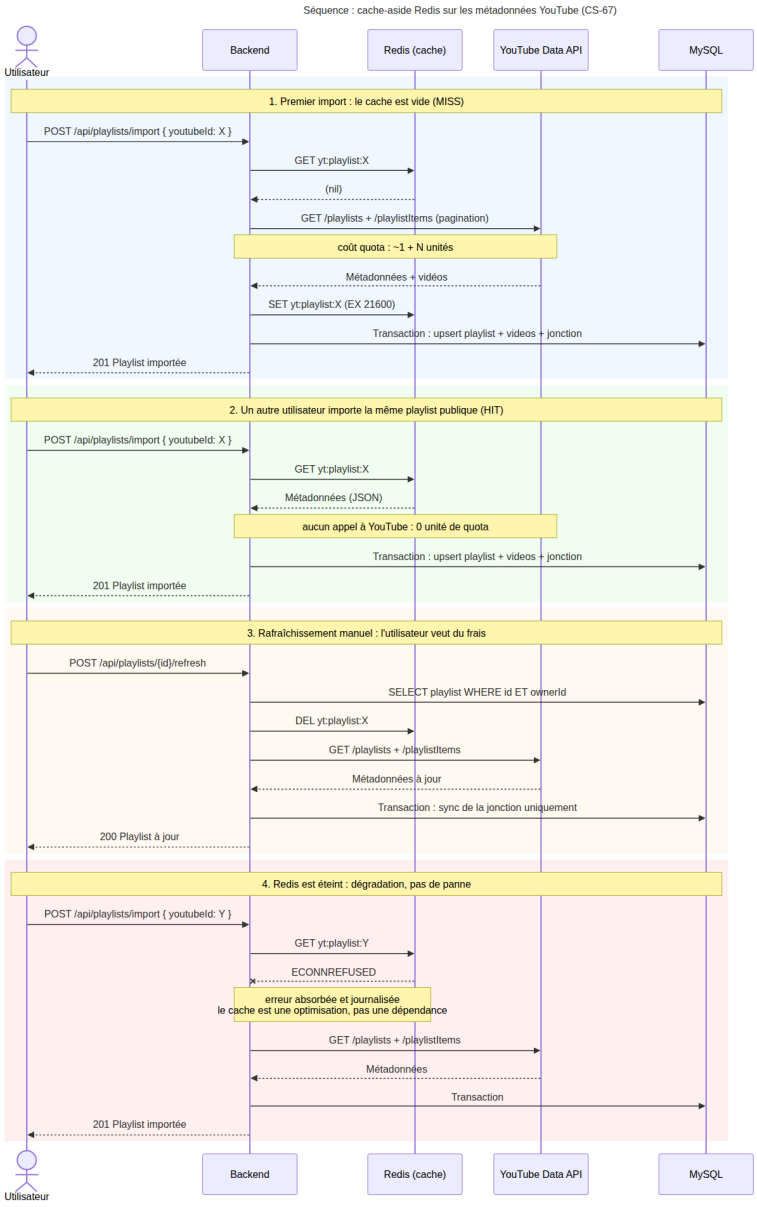


Figure 25. *Séquence du cache-aside Redis (diagramme 14)*

Le second magasin de données est **Redis**, un stockage clé-valeur en mémoire, utilisé comme cache des métadonnées de l'API YouTube (CS-67). Le besoin n'est pas la vitesse mais le **quota** : la YouTube Data API v3 est plafonnée à 10 000 unités par jour et par projet, et quarante utilisateurs qui importent la même playlist publique de 200 vidéos consomment 200 unités pour obtenir exactement les mêmes données.

Le motif retenu est le **cache-aside** : on lit le cache, et seulement en cas d'absence on interroge la source, dont le résultat est mémorisé avec une durée de vie.

```
export async function cacheAside<T>(key: string, loader: () => Promise<T>,
ttlSeconds = 21600) {
  const redis = getClient()

  if (redis) {
    try {
      const hit = await redis.get(key)
```

```
      if (hit !== null) return JSON.parse(hit) as T          // HIT : zéro
appel à YouTube
    } catch (err) {
      logger.warn({ key, err }, 'Lecture du cache impossible')   // panne :
on continue
    }
  }

  const value = await loader()                                 // la source
reste la vérité

  if (redis) {
    try { await redis.set(key, JSON.stringify(value), 'EX', ttlSeconds) }
    catch (err) { logger.warn({ key, err }, 'Écriture du cache impossible') }
  }

  return value
}
```

La règle qui gouverne tout le module : le cache est une optimisation, jamais une dépendance. Redis éteint, saturé ou injoignable, l'application continue de fonctionner en interrogeant directement YouTube. Toutes les erreurs sont absorbées et journalisées, aucune ne remonte à l'utilisateur. Le client est d'ailleurs configuré pour échouer vite plutôt que d'attendre : `connectTimeout` à une seconde, un seul réessai, file d'attente hors ligne désactivée : réessayer indéfiniment transformerait une panne de cache en panne d'API.

Une décision de sécurité, pas de performance, mérite d'être signalée. Les lectures authentifiées, qui peuvent porter sur une playlist **privée**, court-circuitent délibérément le cache :

```
function fetchPlaylistCached(playlistId: string, accessToken?: string) {
  if (accessToken) return fetchPlaylist(playlistId, accessToken)   // privé :
jamais mis en cache
  return cacheAside(playlistKey(playlistId), () => fetchPlaylist(playlistId))
}
```

La clé de cache ne contient que l'identifiant de la playlist, pas celui de l'utilisateur. Mettre en cache une réponse authentifiée sous cette clé reviendrait à servir le contenu d'une playlist privée au premier venu qui en connaîtrait l'identifiant : une fuite de données classique des caches partagés, qui se prévient à la conception de la clé plutôt qu'après coup.

Le rafraîchissement manuel purge la clé avant de relire : la fraîcheur n'est pas devinée par une heuristique, elle est demandée explicitement par l'utilisateur.

Les tests (`server/tests/cache.test.ts`, 9 tests) couvrent le hit, le miss, l'expiration, la sérialisation des structures imbriquées, l'invalidation, et surtout la **dégradation** : lecture, écriture et invalidation en échec doivent renvoyer la bonne valeur sans lever. Tester le comportement en panne autant que le nominal est délibéré : c'est ici la garantie principale, et une garantie non testée n'est qu'une intention.

Mesuré en conditions réelles (26/07/2026, stack de développement) : une playlist publique de 193 vidéos occupe 32 Ko en cache, est servie en **0,59 ms** (p50) là où les 5 appels HTTP séquentiels vers l'API YouTube prennent plusieurs centaines de millisecondes, la pagination interdit de les paralléliser, chaque page fournissant le jeton de la suivante. Trois lectures ont été servies depuis le cache, soit **15 unités de quota économisées** pour une seule playlist et une seule soirée. Le même relevé confirme la décision de sécurité : l'import des playlists personnelles, authentifié, n'a écrit aucune clé dans Redis.

À qui ce cache profite-t-il ? La question mérite d'être posée franchement : pour un utilisateur seul, il n'apporte presque rien, personne n'importe la même playlist deux fois. C'est une optimisation **multi-utilisateurs**. Le cas d'usage visé est celui d'une promotion

suivant le même cours : le premier étudiant qui importe la playlist paie les 5 unités, les suivants sont servis gratuitement. Le gain croît avec le nombre d'utilisateurs partageant des contenus : c'est une propriété du produit autant que du code.

Et ce que le cache ne couvre pas doit être dit aussi : l'import des playlists personnelles, authentifié, le contourne par sécurité et consomme du quota à chaque fois. L'ordre de grandeur reste confortable : une playlist de 100 vidéos coûte 3 unités (1 pour les métadonnées, 2 pour les deux pages de 50), soit environ 3 300 imports authentifiés par jour dans le plafond de 10 000, et une playlist n'est importée qu'une fois puisqu'elle vit ensuite en base. Le risque n'est donc pas l'usage normal mais l'abus, et la parade correcte est une limitation de débit par utilisateur : pas la mise en cache de données privées.

Les deux magasins et la compétence visée. Youcus fait cohabiter un magasin **SQL** (MySQL, données durables, tables et relations, intégrité référentielle) et un magasin **NoSQL** (Redis, couples clé-valeur volatils, sans schéma ni relation). Ce n'est pas une redondance : chacun est choisi pour ce qu'il fait le mieux, et le cache démontre qu'un magasin NoSQL a été retenu là où le relationnel aurait été un mauvais outil, c'est l'objet de la compétence CP8.

Justification complète du choix (Redis contre cache mémoire contre table MySQL), calcul du quota et limites connues : `youcus-docs/14_cache_redis.md`. Diagramme de séquence : `diagrams/14-sequence-cache-redis.png`.

18. Autres composants

La garde d’authentification

```
export function requireAuth(req: Request, res: Response, next: NextFunction):
void {
  const userId = getSessionUserId(req)
  if (!userId) {
    res.status(401).json({ error: 'Authentification requise' })
    return
  }
  setSession(res, userId)   // session glissante : l'expiration recule à
chaque requête
  req.userId = userId
  next()
}
```

Quinze lignes qui portent trois responsabilités : rejeter tôt (avant d’atteindre le moindre service), prolonger la session d’un utilisateur actif, et déposer l’identifiant sur la requête pour que les services n’aient plus jamais à lire le cookie eux-mêmes.

La gestion d’erreurs centralisée

Un seul type d’erreur applicative et un seul point de sortie :

```
export class HttpError extends Error {
  constructor(public status: number, message: string) { super(message) }
}

export function errorHandler(err: unknown, _req, res, _next) {
  const status  = err instanceof HttpError ? err.status : 500
  const message = err instanceof Error ? err.message : 'Erreur interne'
  if (status >= 500) logger.error({ err }, 'Erreur non gérée')
  res.status(status).json({ error: message })
}
```

Le choix de fond est que **toute erreur inattendue devient un `500` générique**. Une erreur Prisma, une exception de parsing, un `undefined` déréférencé : rien ne remonte au client sous sa forme brute. Seules les erreurs que j’ai construites moi-même (`HttpError`) exposent leur message, parce que je l’ai écrit pour être lu par un utilisateur. Les `500` sont journalisés côté serveur avec la pile complète, via `pino`.

La traduction des erreurs de l’API YouTube

Youcus dépend d’une API externe qui peut échouer de plusieurs manières, dont une que l’utilisateur ne peut pas corriger : le dépassement de quota. Les traiter toutes de la même façon produirait des messages incompréhensibles.

```
function mapYouTubeError(status: number, body: unknown): HttpError {
  const message = (body as { error?: { message?:
string } })?.error?.message ?? `statut ${status}`
  if (status === 403 && /quota/i.test(message)) {
    return new HttpError(503, 'Quota YouTube dépassé, réessayez plus tard')
  }
  if (status === 404) return new HttpError(404, 'Playlist introuvable')
  return new HttpError(502, `Erreur de l'API YouTube : ${message}`)
}
```

Le choix des codes est raisonné plutôt que recopié : un quota dépassé devient `503 Service Unavailable` parce que la panne est temporaire et n'est pas la faute du client ; une erreur inattendue de l'API devient `502 Bad Gateway` parce que c'est bien un service en amont qui a échoué, pas Youcus. Un `500` dans ces deux cas serait un mensonge : il désignerait mon serveur comme fautif.

Code cité : `client/src/features/player/FocusPlayer.tsx`, `server/src/services/playlist.service.ts`, `server/src/services/account.service.ts`, `server/src/services/note.service.ts`, `server/src/middleware/requireAuth.ts`, `server/src/middleware/errorHandler.ts`, `server/src/lib/youtube.ts`.

19. Éléments de sécurité

La sécurité de Youcus repose sur un principe simple, appliqué partout : **le serveur ne fait jamais confiance au client**. Toute donnée entrante est validée, toute ressource sortante est filtrée par le propriétaire, et le navigateur n'a jamais accès à ce qui prouve l'identité de l'utilisateur.

L'authentification : aucun mot de passe stocké

Youcus ne gère pas de mots de passe. L'identité est déléguée à Google via OAuth 2.0, ce qui supprime d'un coup toute une famille de risques : pas de base de mots de passe à protéger, pas de hachage à choisir, pas de fuite possible d'identifiants réutilisés ailleurs. La table `User` ne contient qu'un `googleId` et un e-mail.

Le flux OAuth lui-même est protégé contre le **CSRF** par le paramètre `state`, conformément à la spécification :

```
const state = randomUUID()          // valeur imprévisible, générée par
node:crypto
setOAuthState(res, state)           // déposée dans un cookie signé, durée 10
minutes
// ... au retour de Google :
verifyAndClearOAuthState(req, res, received)   // comparée, puis consommée
```

Sans cette vérification, un attaquant pourrait faire aboutir chez la victime un retour d'authentification qu'il aurait lui-même initié, et lier le compte Google de l'attaquant à la session de la victime. Le cookie `state` est à usage unique : il est effacé dès la vérification, réussie ou non.

La session : un cookie signé, invisible au JavaScript

La session est portée par un cookie dont les options constituent l'essentiel de la défense :

```
{
  httpOnly: true,                      // inaccessible à document.cookie
  signed:   true,                      // signature HMAC via SESSION_SECRET
  secure:   isProd,                    // HTTPS obligatoire en production
  sameSite: isProd ? 'none' : 'lax',   // cross-site en prod, front et API
sur deux domaines
}
```

`httpOnly` est la ligne de défense qui compte : même si une faille XSS parvenait à exécuter du script dans la page, ce script ne pourrait pas lire le cookie de session ni l'exfiltrer. `signed` empêche un utilisateur de modifier l'identifiant qu'il contient pour se faire passer pour un autre : la signature ne serait plus valide. Aucun jeton n'est stocké dans `localStorage`, précisément parce que le stockage local est lisible par n'importe quel script de la page.

Le passage à `sameSite: 'none'` en production mérite d'être expliqué, car c'est un affaiblissement apparent : le front est hébergé sur `sevalla.page` et l'API sur `sevalla.app`, donc en contexte *cross-site*. Un cookie `SameSite=Lax` ne serait tout simplement pas envoyé. La contrepartie est que `Secure` devient obligatoire (le cookie ne circule qu'en HTTPS) et que la protection contre le CSRF est alors assurée par le CORS strict décrit plus bas plutôt que par le navigateur.

La session est **glissante** : chaque requête authentifiée repousse l'expiration à sept jours. Un utilisateur actif n'est jamais déconnecté, un utilisateur inactif l'est automatiquement.

Le contrôle d'accès : filtrer par propriétaire, pas vérifier après coup

Le risque le plus courant dans une application de ce type est le *Broken Access Control* : deviner l'identifiant d'une ressource appartenant à quelqu'un d'autre et l'obtenir. J'ai choisi de le rendre structurellement impossible en **intégrant le propriétaire à la requête** plutôt qu'en vérifiant la propriété après lecture :

```
// on ne lit pas la playlist puis on compare son ownerId :
// on demande une playlist QUI APPARTIENT à cet utilisateur.
await prisma.playlist.findFirst({ where: { id, ownerId: userId } })
await prisma.playlist.deleteMany({ where: { id, ownerId: userId } })
```

La différence est décisive. Une vérification après lecture peut être oubliée dans une route sur dix ; un filtre dans la clause `where` ne peut pas l'être sans que la requête cesse de fonctionner. Le même principe descend jusqu'aux ressources indirectes : une note n'est accessible que si la vidéo qu'elle concerne appartient à une playlist de l'utilisateur.

```
where: { id: videoId, playlists: { some: { playlist: { ownerId:
userId } } } }
```

Le middleware `requireAuth` garde toutes les routes protégées et répond `401` sans session valide, avant même d'atteindre le service.

La validation des entrées

Toutes les charges utiles entrantes passent par un schéma **Zod** avec `safeParse`, qui rejette ce qui ne correspond pas au type attendu au lieu de lever une exception. La configuration d'environnement elle-même est validée au démarrage (`server/src/config/env.ts`) : une variable manquante ou malformée fait échouer le lancement du serveur plutôt que de produire un comportement dégradé découvert en production, c'est la leçon tirée de l'incident OAuth de la section 14.

L'accès à la base passe exclusivement par **Prisma**, qui produit des requêtes paramétrées. Aucune concaténation de chaîne SQL n'existe dans le code : le risque d'injection SQL est écarté par construction, pas par vigilance.

Le XSS

Le seul endroit où du contenu écrit par l'utilisateur est transformé en HTML est l'aperçu Markdown des notes. Il est assaini par `rehype-sanitize`, qui retire balises et attributs exécutables avant le rendu (voir l'incident 3 de la section 14). Ailleurs, React échappe automatiquement les valeurs interpolées, et le code ne contient aucun `dangerouslySetInnerHTML`.

Les en-têtes et le CORS

`helmet()` pose les en-têtes de sécurité par défaut : `X-Content-Type-Options`, `X-Frame-Options` (contre le *clickjacking*), `Strict-Transport-Security`, et la suppression de `X-Powered-By` qui révélait la technologie du serveur.

Le CORS est **restreint à une seule origine**, jamais un joker :

```
app.use(cors({ origin: env.CLIENT_ORIGIN, credentials: true }))
```

`credentials: true` est ce qui autorise le cookie de session à traverser ; c'est précisément pour cela que l'origine doit être unique et explicite. Un `origin: '*'` serait d'ailleurs refusé par le navigateur en présence de `credentials`.

Les secrets

Aucun secret n'est versionné : `SESSION_SECRET`, `GOOGLE_CLIENT_SECRET` et l'URL de base de données vivent dans des variables d'environnement, validées au démarrage, et le dépôt ne contient qu'un `.env.example` sans valeurs.

Un incident réel, et ce qu'il a produit. En préparant ce dossier, j'ai constaté qu'une capture d'écran destinée à l'illustrer laissait apparaître le `GOOGLE_CLIENT_SECRET` en clair. J'ai fait tourner le secret dans la console Google Cloud, puis créé **une seconde application Google dédiée à la production**, de sorte que les identifiants de développement et de production soient entièrement distincts. Une fuite sur l'un ne compromet plus l'autre. Un secret exposé se considère comme compromis, indépendamment de qui l'a vu : c'est la rotation qui règle le problème, pas la suppression de la capture.

Le RGPD

Deux droits sont implémentés (CS-22) : l'**export** de toutes les données personnelles au format JSON (`exportUserData`) et la **suppression du compte** (`deleteAccount`). La suppression s'appuie sur les `ON DELETE CASCADE` du schéma décrits à la section 10 : effacer la ligne `User` efface par construction ses playlists, ses notes et sa progression, sans code de nettoyage à maintenir en parallèle et sans risque d'orphelins oubliés.

Limites connues et pistes d'amélioration

Un dossier honnête énonce aussi ce qui n'est pas fait.

- **Les jetons YouTube sont stockés en clair** dans la table `User`. Un accès en lecture à la base donnerait accès aux comptes YouTube des utilisateurs. Le correctif est un chiffrement applicatif au repos (AES-GCM avec une clé en variable d'environnement) : c'est la première amélioration de sécurité que j'apporterais.
- **Aucune limitation de débit** n'est en place. Les routes d'import, qui déclenchent des appels à l'API YouTube, gagneraient à être protégées par `express-rate-limit` : autant contre l'abus que contre l'épuisement du quota d'API.
- **Aucune Content-Security-Policy stricte** : `helmet()` en pose une par défaut, mais l'intégration du lecteur YouTube demanderait de l'ajuster explicitement plutôt que de s'en remettre au réglage générique.

La question du CSRF, puisque le cookie est en SameSite=None

Ce réglage appelle une justification, car il retire la protection que `SameSite` offre par défaut contre les requêtes forgées depuis un autre site. Il est imposé par l'architecture : le front et l'API vivent sur deux domaines distincts, et un cookie en `SameSite=Lax` ne serait tout simplement pas envoyé.

La protection repose alors sur deux mécanismes complémentaires. Le premier est le CORS restreint à l'origine du front, et il faut être précis sur ce qu'il fait : il n'empêche pas un serveur d'exécuter une requête, il empêche un navigateur d'en lire la réponse. Pris seul il ne suffirait pas, et un jury a raison de le relever. Le second ferme réellement la porte : **l'API n'accepte que le type `application/json`**, aucun formulaire encodé n'étant traité. Une requête JSON n'étant pas une requête dite simple, le navigateur la fait précéder d'un appel de contrôle qui échoue pour toute origine tierce ; et les requêtes qu'un site tiers peut envoyer sans ce contrôle sont rejetées faute de corps exploitable.

La protection tient donc par la combinaison des deux. Sa faiblesse est d'être implicite, puisqu'elle dépend de ce que personne n'ajoute un jour un analyseur de formulaires. La rendre explicite demanderait une vérification de l'en-tête `Origin` sur les routes mutatives et, pour un service ouvert au public, un jeton anti-CSRF.

Code : `server/src/app.ts`, `server/src/lib/session.ts`, `server/src/middleware/requireAuth.ts`, `server/src/config/env.ts`, `server/src/services/account.service.ts`, `server/src/routes/auth.route.ts`.

20. Plan de tests et résultats

La stratégie : tester ce qui casse, pas tout

Youcus est un projet mené en solo sur trois semaines. Chercher une couverture exhaustive aurait consommé le temps de développement sans réduire le risque là où il est réel. J'ai donc choisi une stratégie explicite : **couvrir en priorité ce dont la casse est silencieuse ou coûteuse**, et assumer de laisser peu couvert ce dont la casse est immédiatement visible.

Concrètement, trois catégories sont testées en priorité :

1. **Ce qui détruit des données de l'utilisateur.** Le bug de cascade (section 14) a effacé des notes et des progressions à chaque rafraîchissement sans qu'aucune erreur ne s'affiche. C'est le type de défaut qu'un test de non-régression doit verrouiller définitivement.
2. **Ce qui garde les données d'un utilisateur séparées de celles d'un autre.** Le contrôle d'accès par `ownerId` (section 19) est la garantie de sécurité principale ; elle doit être prouvée, pas supposée.
3. **Ce qui doit continuer de marcher quand une dépendance tombe.** Redis éteint, quota YouTube dépassé, API en panne : le comportement dégradé est une garantie, et une garantie non testée n'est qu'une intention.

À l'inverse, une erreur de mise en page ou un libellé mal traduit se voit à l'écran en une seconde : y consacrer des tests aurait été un mauvais emploi du temps.

Les outils

Vitest pour les deux moitiés du monorepo, avec **Testing Library** côté client (on interroge le DOM comme le ferait un utilisateur : par rôle et par texte, jamais par classe CSS) et **Supertest** côté serveur pour les tests d'API de bout en bout. Les dépendances externes (Prisma, l'API YouTube, Redis) sont **simulées** : les tests doivent tourner en intégration continue sans base de données, sans réseau et sans quota.

Résultats réels (relevés le 08/08/2026)

```
Serveur : 15 fichiers, 56 tests — 100 % au vert
Client  : 15 fichiers, 25 tests — 100 % au vert
Total   : 30 fichiers, 81 tests
```

	LIGNES	BRANCHES	FONCTIONS
Serveur	70,1 %	83,4 %	70,9 %
Client	71,4 %	77,3 %	74,3 %

La couverture est mesurée par `@vitest/coverage-v8`, configurée dans ce projet et lançable par `npm run test:coverage`.

Le chiffre à commenter n'est pas le pourcentage global, c'est sa répartition. Une couverture uniforme de 70 % serait un mauvais signe : elle voudrait dire que le code critique et le code trivial sont testés au même degré. Ce n'est pas le cas :

MODULE	LIGNES COUVERTES	POURQUOI
`lib/cache.ts`	**98,6 %**	Le comportement en panne est la garantie principale
`middleware/requireAuth.ts`	**100 %**	Porte d'entrée de toutes les routes protégées
`services/account.service.ts`	**96,6 %**	Export et suppression RGPD : irréversible
`services/note.service.ts`	88,2 %	Contrôle d'accès indirect (note vers vidéo vers playlist)
`services/playlist.service.ts`	71,8 %	Cœur métier, chemins d'erreur partiellement couverts
`lib/googleOAuth.ts`	**7,6 %**	Assumé : demanderait de simuler le serveur d'autorisation Google

Côté client, les composants portant de la logique sont bien couverts (`SettingsPage` 96,6 %, `LandingPage` et `LoginPage` 100 %) tandis que les pages qui ne font qu'assembler des composants déjà testés le sont peu.

Le test qui compte le plus

`server/tests/playlist.refresh.test.ts` verrouille le bug de cascade de la section 14 : il vérifie qu'**une note et une progression survivent à un rafraîchissement de playlist**. Ce test aurait échoué sur le code d'avant le refactor. Il est aujourd'hui la preuve que la régression ne peut pas revenir sans être détectée.

Dans le même esprit, quatre des neuf tests du cache portent uniquement sur la **panne** : lecture impossible, écriture impossible, invalidation impossible, chacune devant renvoyer la bonne valeur sans lever d'erreur.

L'automatisation

Le retour arrière. La chaîne de livraison publie chaque service sous deux étiquettes : `latest`, que la production suit, et l'empreinte exacte du commit. Chaque déploiement laisse donc derrière lui une image immuable et identifiable, et revenir à l'état précédent consiste à redéployer l'image portant l'empreinte du commit voulu, sans reconstruction ni dépendance à l'état du dépôt. La limite tient aux migrations de base : une migration appliquée n'est pas annulée par ce retour, ce qui impose de n'écrire que des migrations compatibles avec la version précédente, et c'est une des raisons pour lesquelles la conclusion place l'environnement de préproduction en tête des priorités.

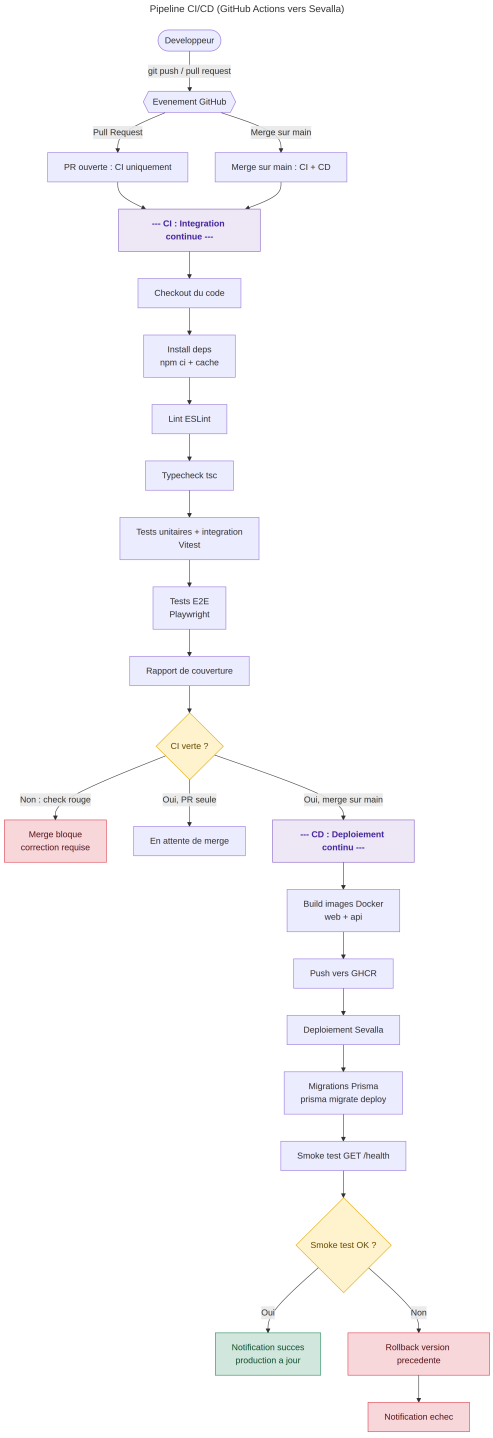


Figure 26. *Le pipeline CI/CD (diagramme 12)*

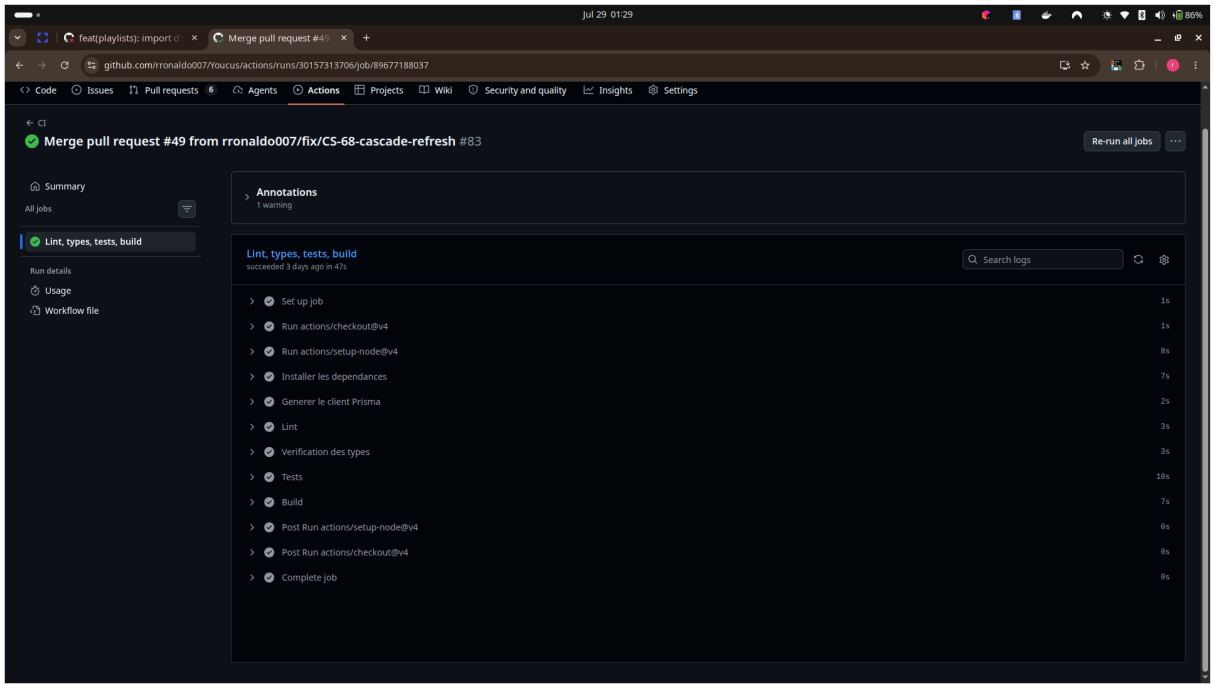


Figure 27. *GitHub Actions : un run vert complet sur `main` : les douze étapes passent en 47 s, du lint au build*

Les tests tournent à chaque *push* et à chaque *pull request* via **GitHub Actions** (`.github/workflows/`), dans un travail unique qui enchaîne **lint vers vérification des types vers tests vers build**. L'ordre est délibéré : les vérifications les moins coûteuses échouent en premier, et le *build* ne s'exécute que sur du code déjà validé. Le pipeline a rempli son rôle dès le premier jour : le 19/07 à 13h56, une CI rouge a arrêté une fusion (section 4). C'est la démonstration la plus directe de son utilité : un pipeline qui ne bloque jamais rien ne prouve rien.

Limites assumées

- **Aucun test de bout en bout dans un vrai navigateur.** Un parcours complet (se connecter, importer, lire, annoter) n'est vérifié que par morceaux. Playwright serait l'outil ; c'est la première amélioration que j'apporterais au plan de tests.
- **Le flux OAuth n'est pas testé**, faute d'un serveur d'autorisation simulé. C'est la zone la moins couverte (7,6 %) et je le sais.
- **Aucun test de charge.** L'application n'a jamais été mesurée au-delà d'un utilisateur ; les chiffres de quota de la section 17 sont des calculs, pas des relevés sous charge.

21. Jeu d’essai de la fonctionnalité la plus représentative : l’import d’une playlist

J’ai choisi l’import d’une playlist comme fonctionnalité de référence parce qu’elle traverse toutes les couches de l’application (l’interface, le service métier, l’API YouTube paginée, la transaction Prisma) et parce que c’est elle qui a provoqué le refactor le plus important du projet (section 14, section 16). Le jeu d’essai a été déroulé le 27/07/2026 sur la stack de développement, captures et relevés SQL à l’appui.

Données en entrée

ÉLÉMENT	VALEUR
Entrée utilisateur	URL publique `https://www.youtube.com/playlist?list=PLybg94GvOJ9FoGQeUMFZ4SWZsr30jlUYK` : « Mathematics (All Of It) », chaîne Professor Dave Explains
État initial du compte	Aucune playlist
État initial de la base	205 vidéos canoniques issues d’imports antérieurs : condition conservée volontairement pour éprouver l’appariement par `youtubeId`
Donnée témoin	Une note Markdown rédigée la veille (26/07, 19h31) sur la vidéo « Introduction to Mathematics », conservée en base alors que la playlist avait été supprimée

Données attendues

- YouTube annonce **193 vidéos** pour cette playlist *(capture de la page YouTube)* ;
- 193 vidéos importées, dans l’ordre de la playlist, numérotées 1 à 193 dans l’interface (positions 0 à 192 en base) ;
- l’API étant paginée à 50 éléments, **4 appels** successifs à `playlistItems` (50 + 50 + 50 + 43), non parallélisables puisque chaque page fournit le jeton de la suivante ; avec l’appel `playlists` qui lit les métadonnées de la playlist, l’import coûte **5 unités de quota** au total (section 17) ;
- **aucune vidéo créée en double** : les 193 vidéos existant déjà en base doivent être reconnues par leur `youtubeId`, pas recréées ;
- la note témoin doit **réapparaître** sur la vidéo 1, sans intervention.

Données obtenues (relevé du 27/07/2026)

- Message de l’interface : **« Playlist « Mathematics (All Of It) » importée (193 vidéos). »**, environ quatre secondes après le clic (horodatage des captures : 11:56:09 vers 11:56:13).
- Relevé SQL sur la base : **193 lignes** dans la table de jonction, positions **0 à 192, 193 positions distinctes**, aucune vidéo manquante, aucun doublon de position.
- Compte de la table `Video` avant et après l’import : **205 vers 205**. Zéro ligne créée : l’upsert a reconnu les 193 vidéos par leur `youtubeId`.

- La note témoin est **revenue d'elle-même** sur « Introduction to Mathematics », rendu Markdown assaini dans l'aperçu, horodatage d'origine intact (26/07, 19h31).
- Après lecture et « Marquer comme vue » sur la première vidéo, la progression de la playlist affiche 1 % : le suivi repart sur la playlist réimportée.

Analyse des écarts

La durée des vidéos est absente (0 seconde partout). C'est l'écart principal, et il est structurel : l'import n'interroge `playlistItems` qu'avec `part=snippet`, qui ne contient pas la durée. La renseigner demanderait un appel supplémentaire `videos?part=contentDetails` par lot de 50, soit 4 unités de quota de plus pour une playlist de cette taille. Le champ `durationSeconds` existe déjà dans le schéma et dans l'API de lecture : le correctif est identifié, je l'ai laissé hors périmètre en connaissance de cause.

Annoncé et importé sont identiques ici (193 / 193) parce que cette playlist ne contient ni vidéo privée ni vidéo supprimée. Sur une playlist qui en contient, l'API les exclut silencieusement : un écart entre le nombre annoncé par YouTube et le nombre importé est donc un comportement attendu et documenté, pas une anomalie.

Le retour de la note n'était pas garanti par l'ancienne version du code. Avant le refactor N:N (section 14), la suppression de la playlist aurait entraîné les vidéos dans sa cascade et détruit la note. Ce jeu d'essai rejoue donc, en conditions réelles, le scénario exact du bug corrigé, et le correctif tient.

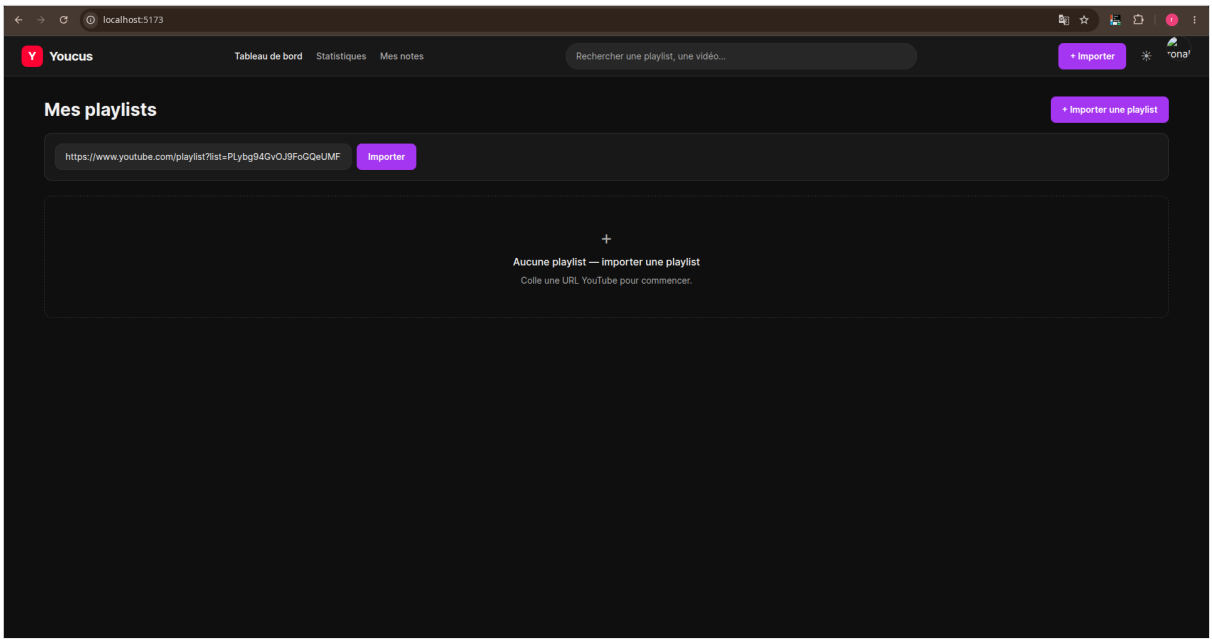


Figure 28. *État initial : aucune playlist, URL collée dans le champ d'import (c21a)*

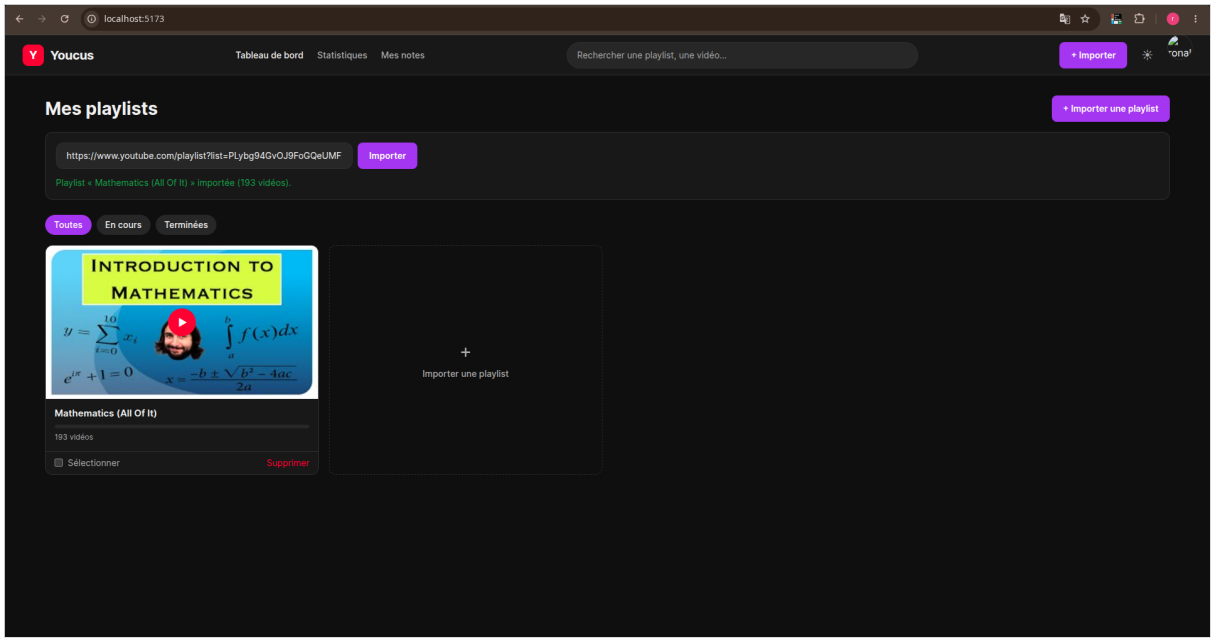


Figure 29. *Le résultat : « Mathematics (All Of It) » importée, 193 vidéos (c21c)*

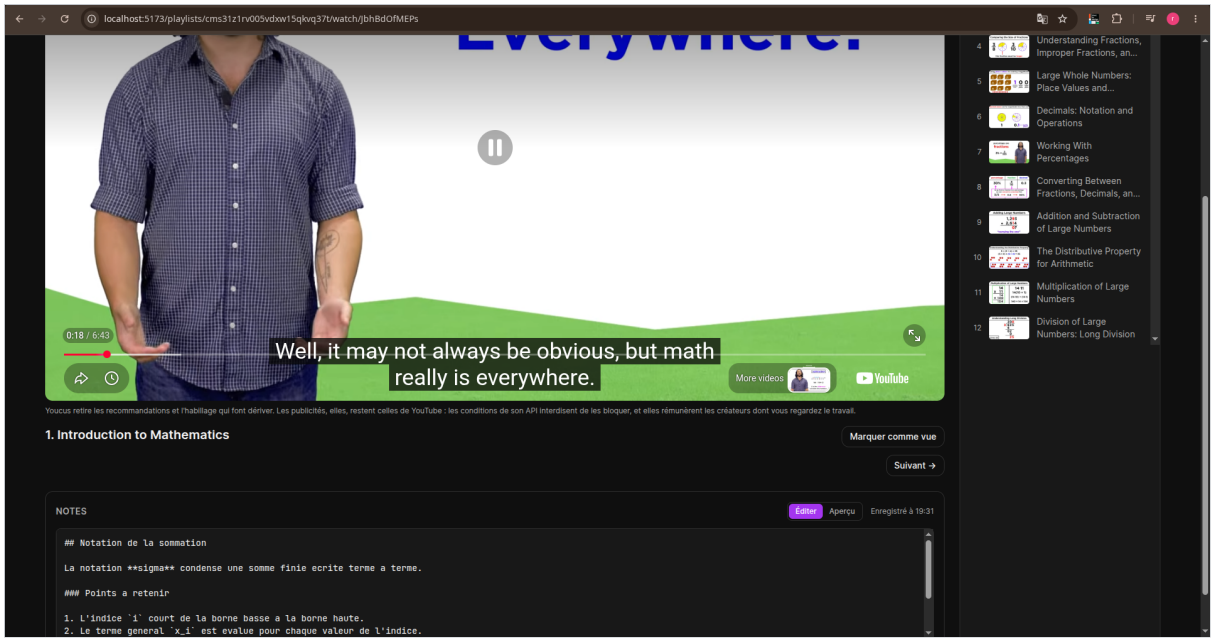


Figure 30. *La note témoin revenue sur la vidéo 1, rendu Markdown assaini, horodatage intact (c21d)*

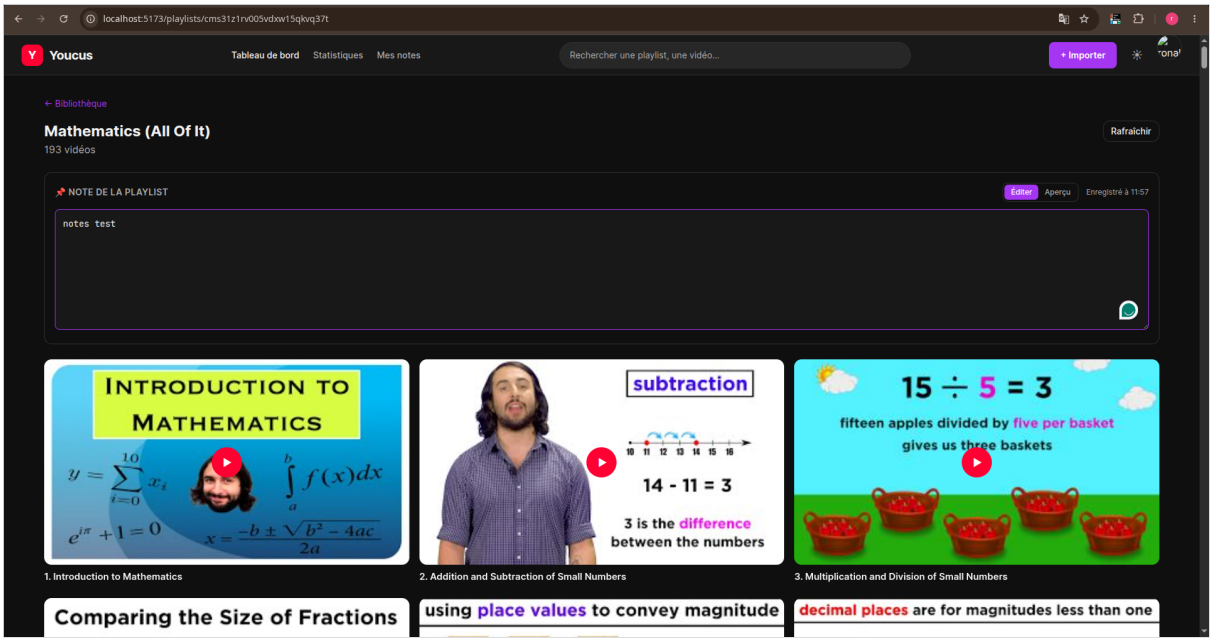


Figure 31. *Détail de la playlist importée : 193 vidéos ordonnées (c21g)*

22. Veille sur les vulnérabilités de sécurité

Comment la veille est organisée

Ma veille repose sur trois canaux complémentaires, du plus automatisé au plus humain.

Une veille outillée, branchée sur le dépôt. J'ai configuré Dependabot dès la mise en place du socle technique (ticket CS-50) : contrôle hebdomadaire des dépendances npm et des actions GitHub du pipeline, avec un plafond de cinq pull requests ouvertes par écosystème (npm d'un côté, actions GitHub de l'autre). Ce choix d'outil n'est pas décoratif : la semaine du 18 juillet, Dependabot a ouvert neuf pull requests, dont des montées de version majeures, Prisma 5.22 vers 7.8, `actions/setup-node` 4 vers 7. Chaque PR passe par la CI (lint, types, tests, build), ce qui donne un premier verdict automatique sur la compatibilité.

Ces neuf PR n'ont volontairement pas été fusionnées à l'aveugle, et la raison tient en une distinction : **un correctif de sécurité se traite immédiatement, une montée de version se planifie.** Aucune de ces PR n'est un avis de sécurité, ce sont des montées de routine, dont des majeures. Prisma 5 vers 7, c'est deux versions majeures d'un coup sur l'ORM, la couche qui touche toutes les données de l'application ; une majeure annonce par définition des ruptures de compatibilité.

La fusionner en plein sprint de livraison serait un risque de casse maximal pour un gain fonctionnel nul. Le traitement prévu : lire les notes de version, monter en local, rejouer les tests et les migrations sur une copie de la base, puis fusionner. Si l'une de ces PR portait une CVE, elle passerait devant tout le reste : c'est exactement pour faire ce tri que la veille existe.

Une veille au moment de construire. Pour chaque brique sensible du projet, j'ai creusé la documentation de référence avant d'implémenter, plutôt que de copier une solution toute faite. L'exemple le plus formateur est celui des cookies : construire la session m'a demandé de comprendre réellement les attributs `HttpOnly` (le jeton invisible au JavaScript, donc hors de portée d'un XSS), `Secure`, `SameSite`, et la règle du rattachement au domaine, que la topologie front statique / API séparée m'a fait apprendre à mes dépens : d'abord l'incident d'adresses OAuth de la section 14, puis le passage à `SameSite=None` en production, qui n'est pas un choix par défaut mais une conséquence assumée de l'architecture, documentée à la section 19. C'est cette recherche qui a produit les protections de la section 19, du cookie `state` anti-CSRF à l'assainissement du Markdown.

Le socle de la formation. Les modules sécurité du parcours iSCOD m'ont donné la grille de lecture des failles classiques (injection, XSS, CSRF, exposition de secrets) que j'applique ensuite aux cas concrets du projet.

Vulnérabilités trouvées et corrigées pendant le projet

Un XSS potentiel dans le rendu des notes. Les notes sont rédigées en Markdown et rendues en HTML : sans assainissement, une note contenant du HTML malveillant s'exécuterait dans le navigateur. La faille a été identifiée et fermée le jour même de la construction de l'aperçu (CS-17, 20/07) : le rendu passe par `rehype-sanitize`, qui filtre le HTML généré sur liste blanche. La section 14 raconte l'incident complet, dépendance oubliée comprise.

Un secret OAuth exposé dans une capture d'écran. En préparant ce dossier, j'ai constaté qu'une capture destinée à illustrer la configuration Google Cloud laissait apparaître le

`GOOGLE_CLIENT_SECRET` en clair. Un secret exposé se considère comme compromis, quel que soit le nombre de personnes qui l'ont vu : j'ai fait tourner le secret dans la console Google, puis créé une seconde application Google dédiée à la production, de sorte que les identifiants de développement et de production soient entièrement séparés, une fuite sur l'un ne compromet plus l'autre (section 19).

Ce que cette veille n'attrape pas encore

Dependabot surveille les versions publiées, pas les avis de sécurité en temps réel : une CVE critique publiée entre deux contrôles hebdomadaires peut attendre jusqu'à sept jours. L'étape suivante serait d'activer les alertes de sécurité GitHub et un `npm audit` en CI, pour réagir à l'avis plutôt qu'à la version. C'est le prolongement naturel du dispositif actuel, identifié mais pas encore en place.

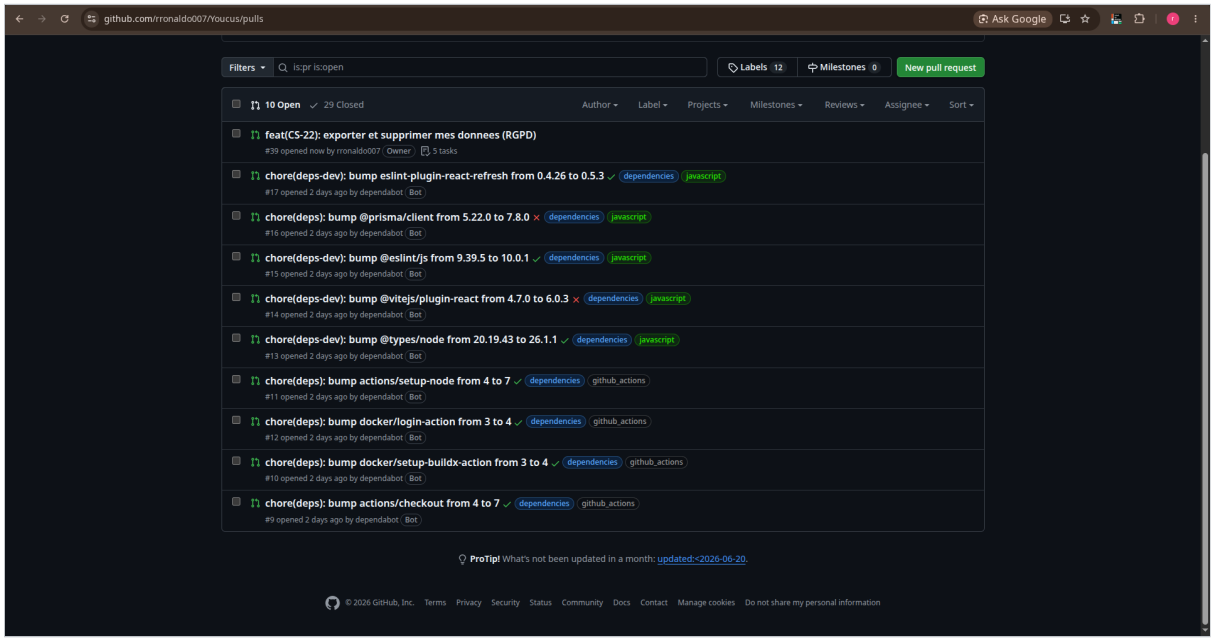


Figure 32. *GitHub : les neuf pull requests Dependabot du cycle du 18/07 (capture du 20/07)*

23. Conclusion

Youcus est en ligne. Ce qui n'était en juin qu'un constat, celui d'une bibliothèque éducative considérable enfermée dans un environnement conçu pour capter l'attention, est devenu une application déployée : on s'y connecte par son compte Google, on y importe ses playlists, on les suit dans un lecteur débarrassé des recommandations, on y prend des notes rattachées au moment exact de la vidéo, et sa progression est conservée.

Le projet couvre l'ensemble de la chaîne que le titre demande de démontrer. L'expression des besoins et les maquettes ont précédé le code ; la conception de la base a été menée en Merise puis en modèle entités-relations ; le développement s'est fait ticket par ticket sur un backlog Jira, avec une branche et une pull request pour chacun ; une intégration continue vérifiait le style, les types, les tests et la construction à chaque proposition de fusion, et un déploiement automatique portait en production ce qui passait. Les onze compétences du référentiel sont mises en œuvre, et la section 1 indique pour chacune où en trouver la preuve.

Je le présente pour ce qu'il est, sans l'agrandir. Youcus est un projet personnel qui a servi de terrain de démonstration, et non un produit exploité : il est déployé, mais il n'a pas encore d'utilisateurs autres que moi, ce que confirme sa facture d'hébergement de moins de quatre dollars sur un mois complet. Ses limites techniques sont nommées une à une dans les sections consacrées à la sécurité, aux tests et au jeu d'essai, avec pour chacune le correctif que j'ai identifié. Ce que ce projet établit, ce n'est pas le succès d'un produit, c'est ma capacité à conduire une application de l'analyse du besoin jusqu'à la production, et à rendre compte de ce que j'ai fait comme de ce que je n'ai pas fait.

Ce que ce projet m'a appris

Les deux difficultés qui m'ont le plus coûté n'étaient pas des difficultés de code. L'une était un écart entre ma conception et mon implémentation, l'autre un critère d'acceptation qui n'avait jamais été écrit. Toutes deux se sont déclarées tard, sous la forme d'un seul bug qui détruisait les données des utilisateurs, et toutes deux étaient déjà présentes le jour où j'ai commencé à coder. Un modèle de données mal tranché ou une story incomplète ne se rattrapent pas au clavier : ils attendent, et ils se paient au moment le plus coûteux.

Cela a changé ma façon de lire un ticket. La story de synchronisation prévoyait un critère sur les notes, mais uniquement pour les vidéos retirées par le créateur de la playlist. Le cas rare était spécifié, le cas ordinaire ne l'était pas, et c'est le cas ordinaire qui a détruit les données. Je relis désormais des critères d'acceptation en cherchant ce qu'ils ne disent pas.

Cela a aussi changé ce que j'attends d'un test. La fonctionnalité fautive était livrée avec trois tests qui passaient, dont un vérifiait que la synchronisation remplaçait bien les vidéos : il validait le mécanisme même qui effaçait les notes. Un test qui contrôle ce qu'une opération produit, sans contrôler ce qu'elle doit laisser intact, donne une confiance qu'il n'a pas méritée.

L'architecte plutôt que le producteur de lignes

Cette leçon a pris une portée qui dépasse le projet. J'ai développé Youcus en m'appuyant sur une IA générative, ce que ce dossier déclare en page 2, et cette expérience m'a convaincu d'une chose : plus la production de code devient rapide et peu coûteuse, plus la

valeur se déplace vers ce qui la précède. Décider d'un modèle de données, arbitrer une architecture, découper un besoin en tickets dont les critères couvrent le cas ordinaire autant que le cas rare : rien de tout cela ne s'accélère, et tout cela détermine ce que le code produit vaudra. Le bug qui m'a le plus coûté en est la démonstration : aucune vitesse d'écriture n'aurait compensé un modèle tranché trop vite, et je ne l'ai réglé qu'en revenant à ma conception initiale puis en écrivant à la main une migration que l'outil aurait mal générée. Deux décisions prises contre la solution rapide.

C'est le rôle vers lequel je veux aller, et ce projet m'a servi à en éprouver l'exigence : celui qui conçoit, arbitre et vérifie, plutôt que celui qui produit des lignes. Mon alternance et mon projet personnel m'ont d'ailleurs fait vivre deux régimes de contrôle opposés. Chez mon employeur, aucun code ne part en production sans une revue humaine, puis sans un environnement intermédiaire où l'on vérifie sur des données réalistes ce que les tests ne voient pas. Sur Youcus, seul, j'ai travaillé sans relecteur et sans palier : une fois la pull request fusionnée, le déploiement partait en production, et l'intégration continue était mon unique filet. Elle a tenu, et ce dossier en donne la trace. Mais un test ne vérifie que ce que j'ai pensé à vérifier, là où une revue et une préproduction attrapent ce à quoi je n'ai pas pensé, c'est-à-dire précisément la catégorie de problèmes qui m'a coûté le plus cher.

Perspectives

Je ne compte pas laisser Youcus à l'état où ce dossier le décrit. Le projet continue, et la première chose que j'y ajouterai n'est pas une fonctionnalité : c'est un environnement de préproduction, avec la procédure de retour arrière qui va avec.

Ce choix peut surprendre alors que ce dossier nomme des limites plus visibles, à commencer par les jetons YouTube stockés en clair. Il est pourtant l'ordre le plus sûr, et c'est mon propre projet qui me l'a appris : chiffrer ces jetons suppose de transformer des données déjà en base, c'est-à-dire exactement le type d'opération qui a détruit des notes d'utilisateur quand je l'ai conduite sans palier intermédiaire. Viennent ensuite le chiffrement des jetons OAuth au repos, la limitation de débit sur les routes qui appellent l'API YouTube, et des tests de bout en bout dans un navigateur, qui couvriraient le parcours d'authentification que mes tests actuels ne peuvent pas atteindre. Chacune de ces limites est décrite à l'endroit du dossier qui la concerne ; ce que la préproduction apporte, c'est le moyen de les livrer sans reproduire l'erreur qui m'a le plus coûté.

Annexes

Annexe A : MCD de conception initiale

Le modèle imaginé en conception (8 tables), avant la simplification à 5 tables puis le retour au N:N décrits à la section 9.

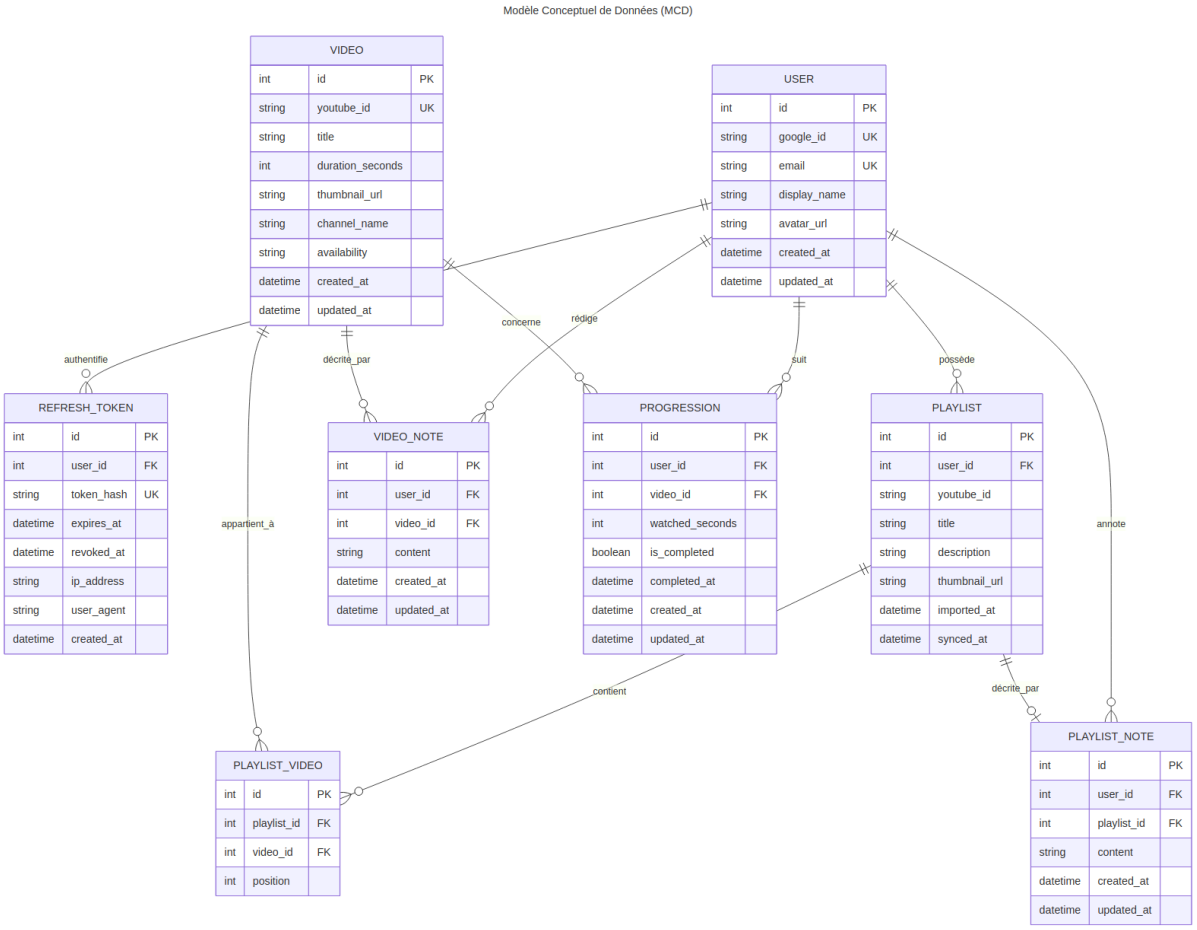


Figure 33. *MCD de conception initiale*

Annexe B : Wireframes et maquettes (sélection)

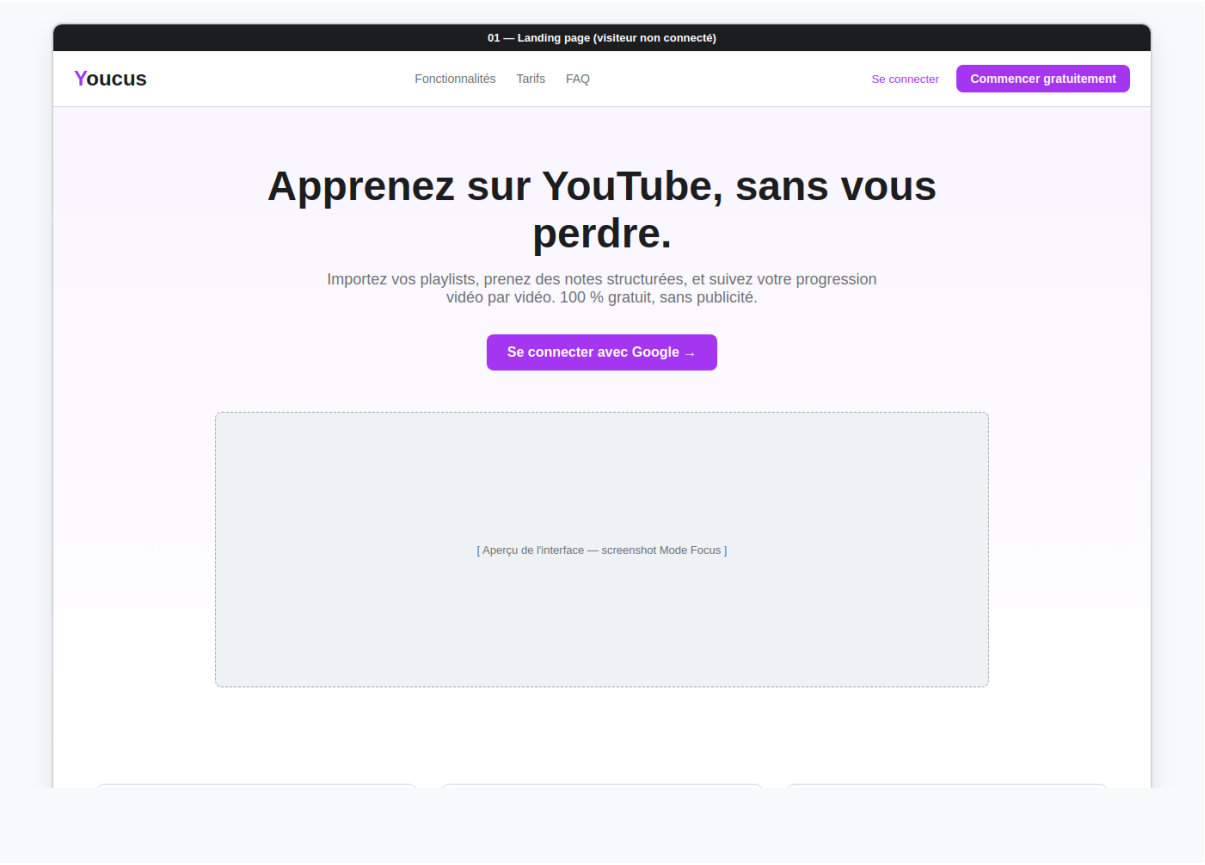


Figure 34. *Landing et connexion*

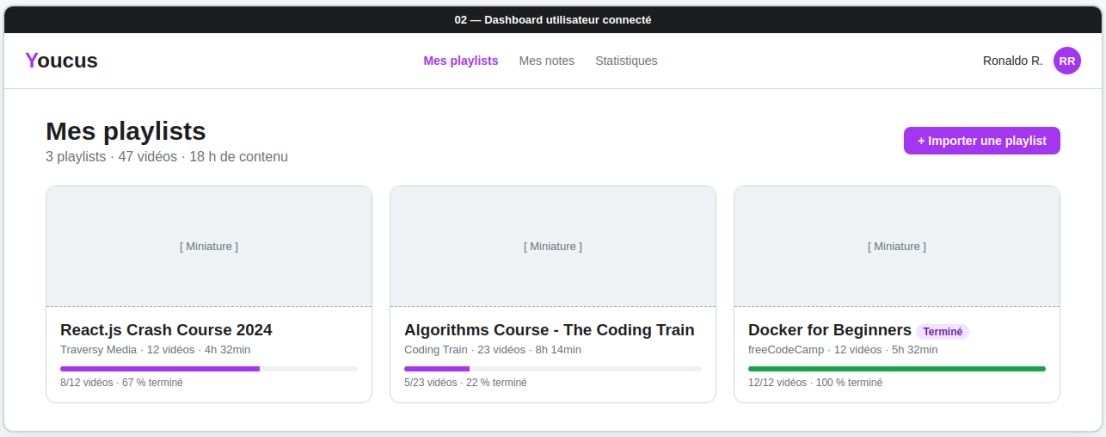


Figure 35. *Dashboard*

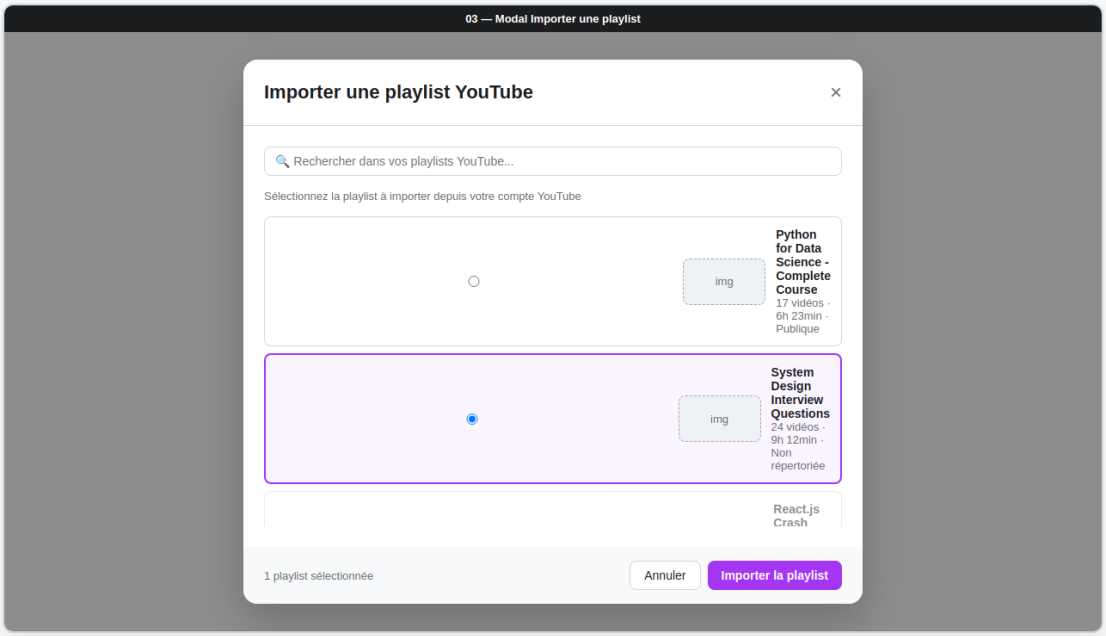


Figure 36. *Modale d'import*

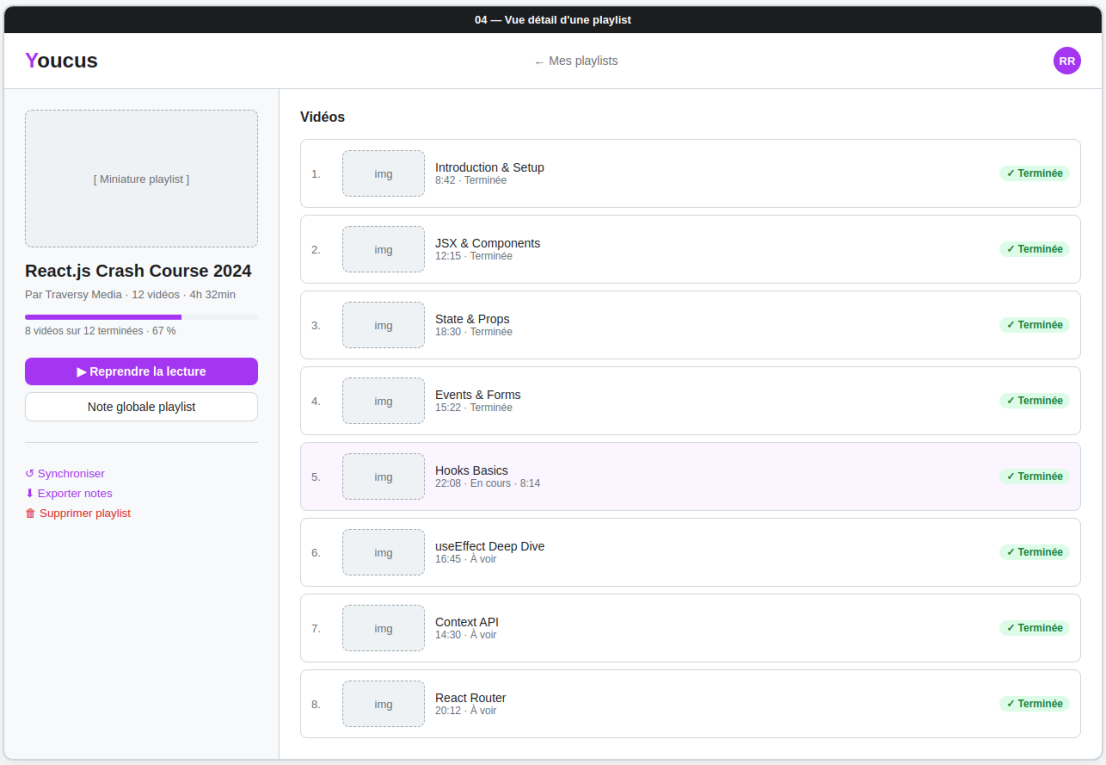


Figure 37. *Détail d'une playlist*

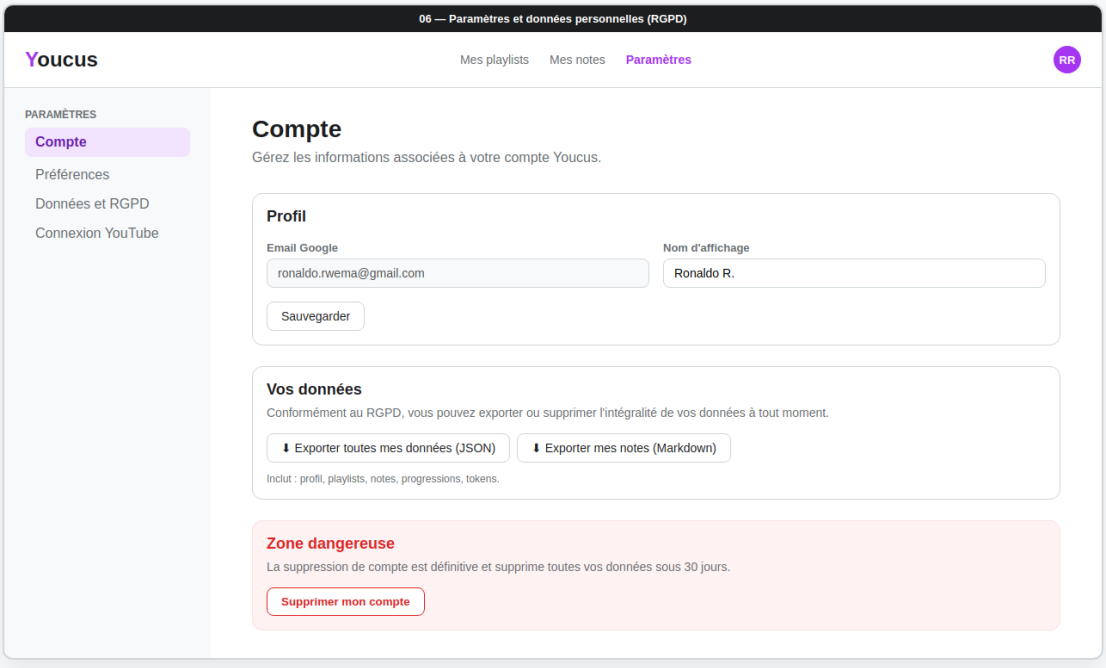


Figure 38. *Compte et données RGPD*

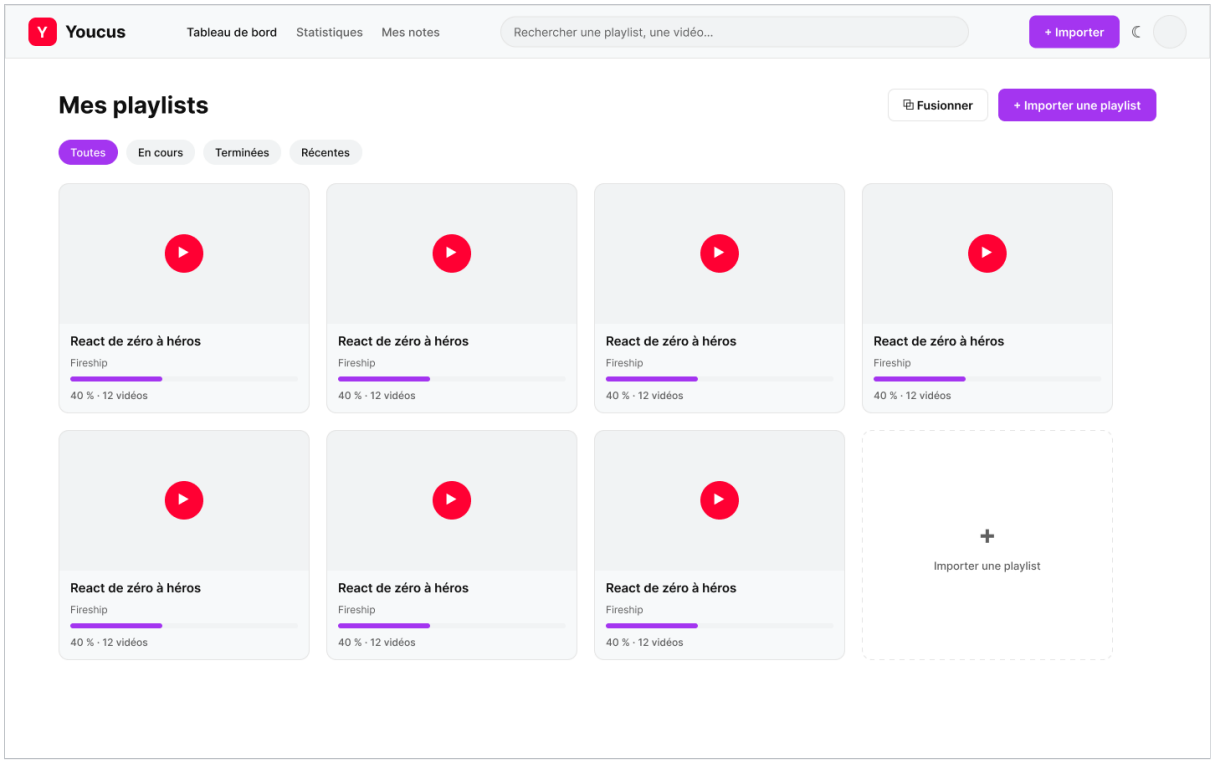


Figure 39. *Maquette Figma : le dashboard (haute fidélité)*

Annexe C : Diagrammes et captures complémentaires

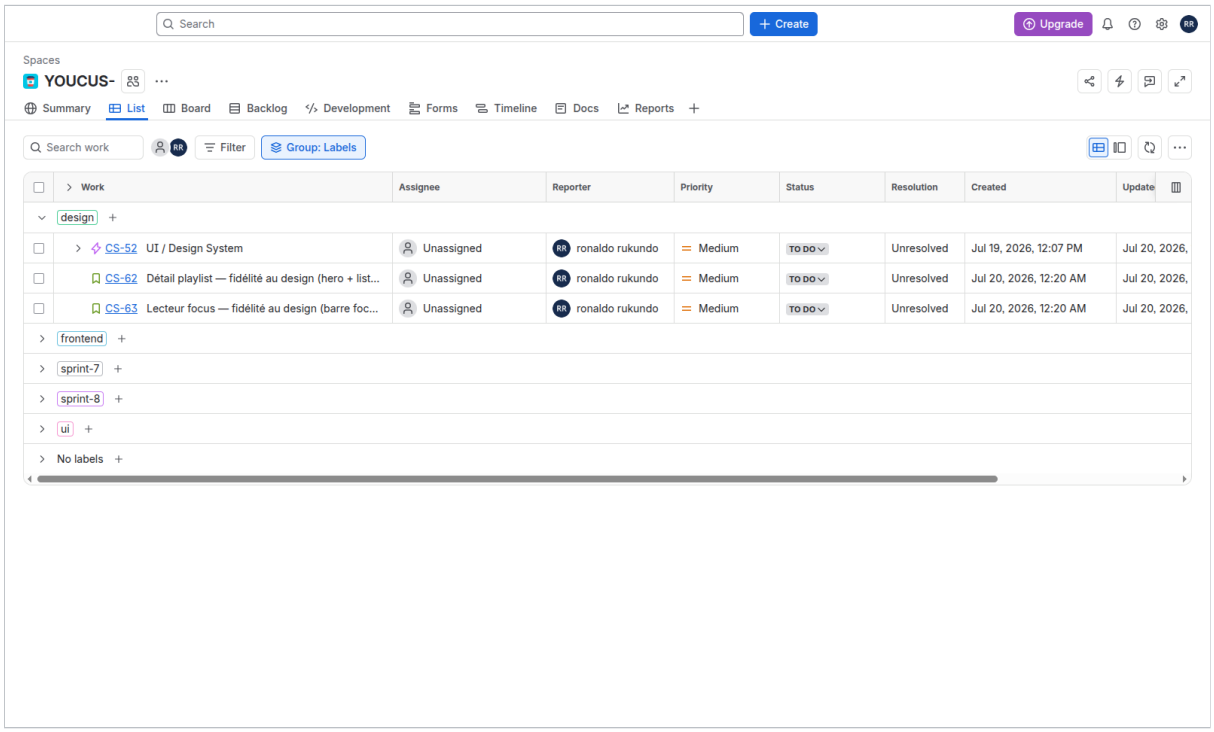


Figure 40. *Jira : la liste des tickets du projet*

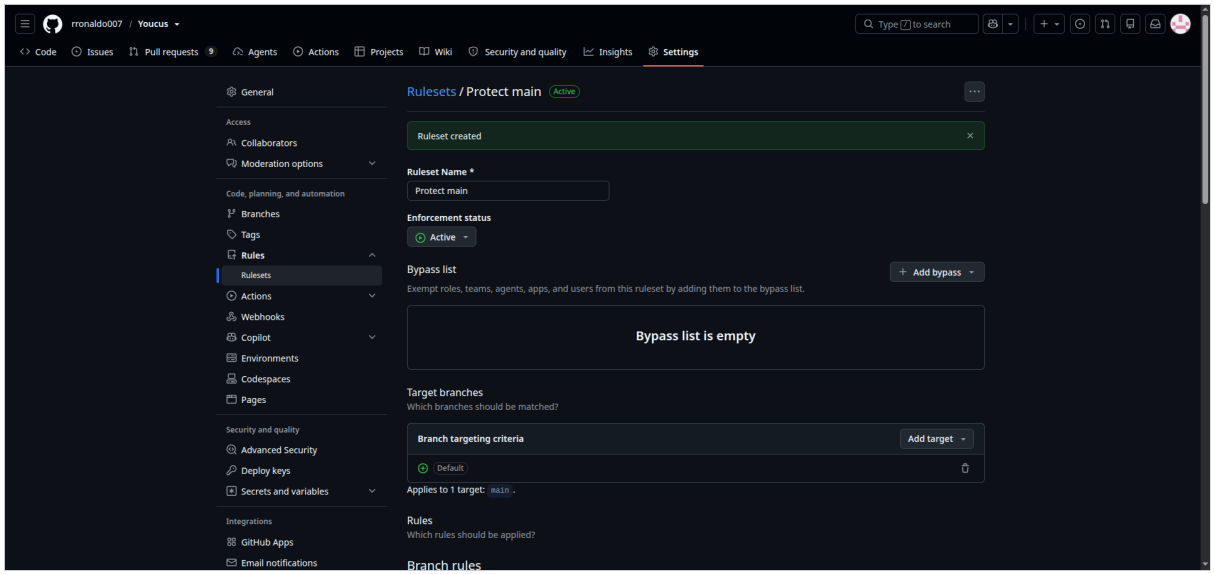


Figure 41. *GitHub : protection de la branche main (ruleset)*

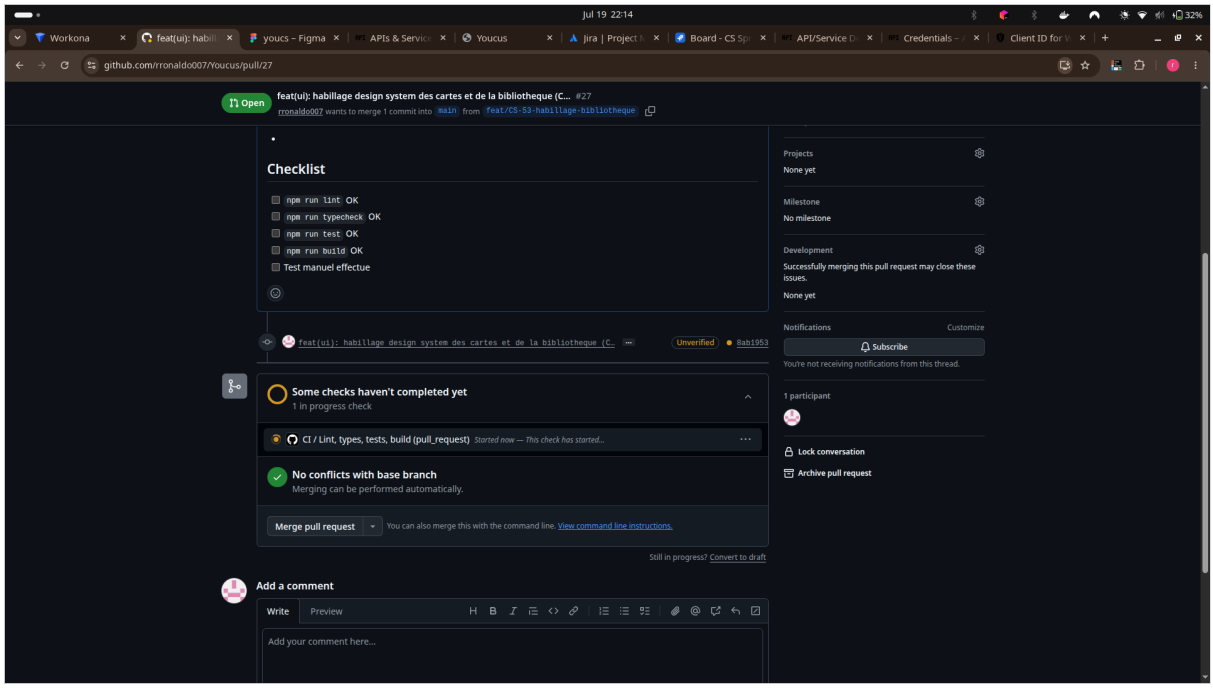


Figure 42. *GitHub : une pull request avec sa checklist et ses vérifications CI*

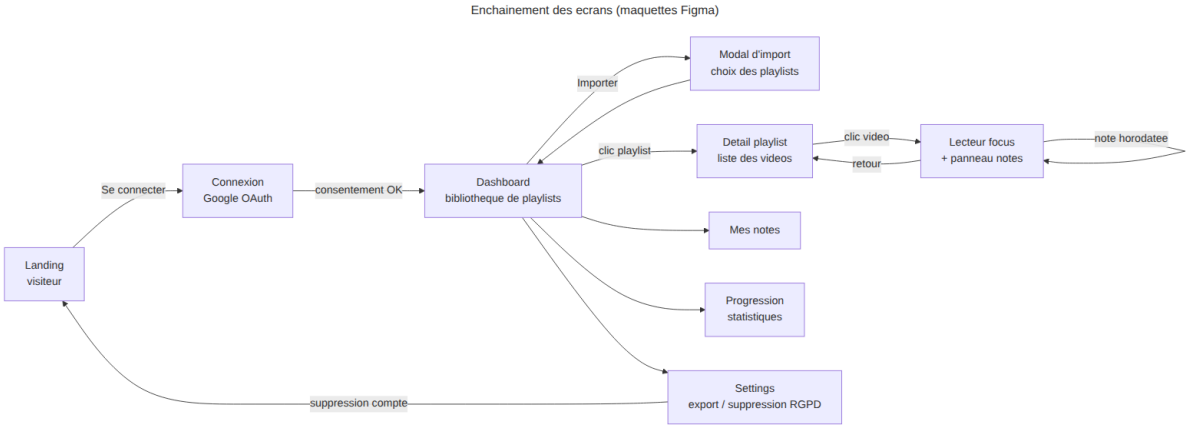


Figure 43. *Enchaînement des écrans (diagramme 13)*

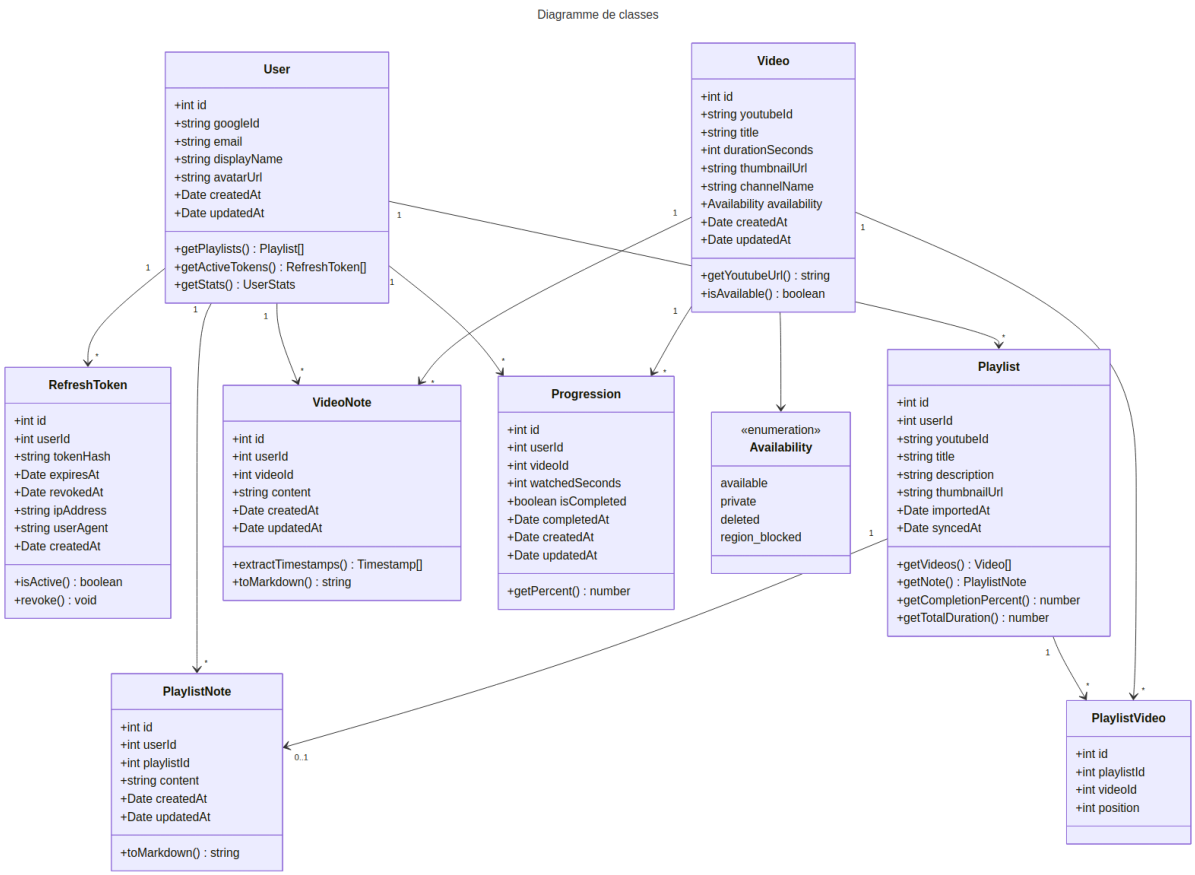


Figure 44. *Diagramme de classes*

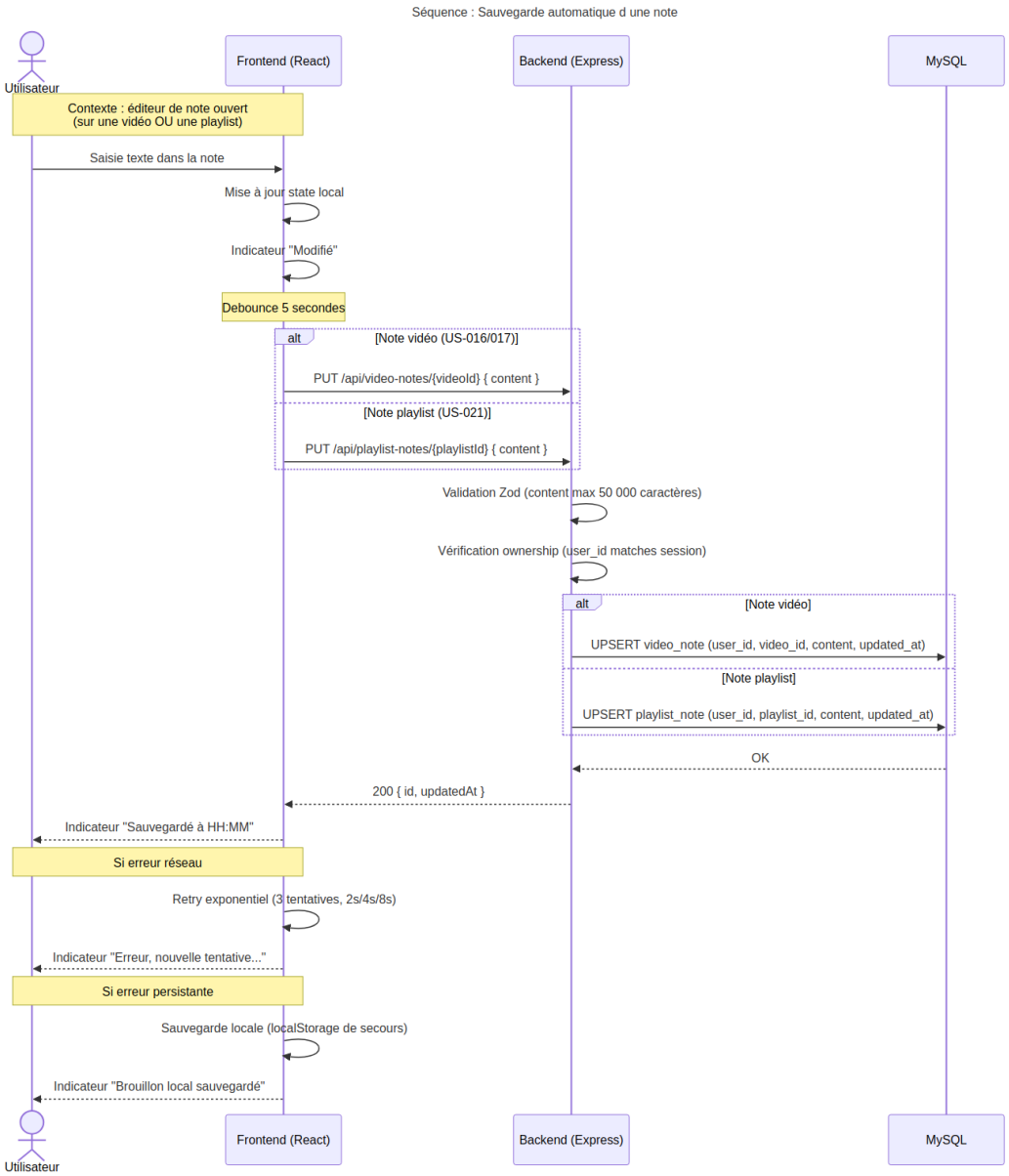


Figure 45. *Séquence : sauvegarde d'une note*

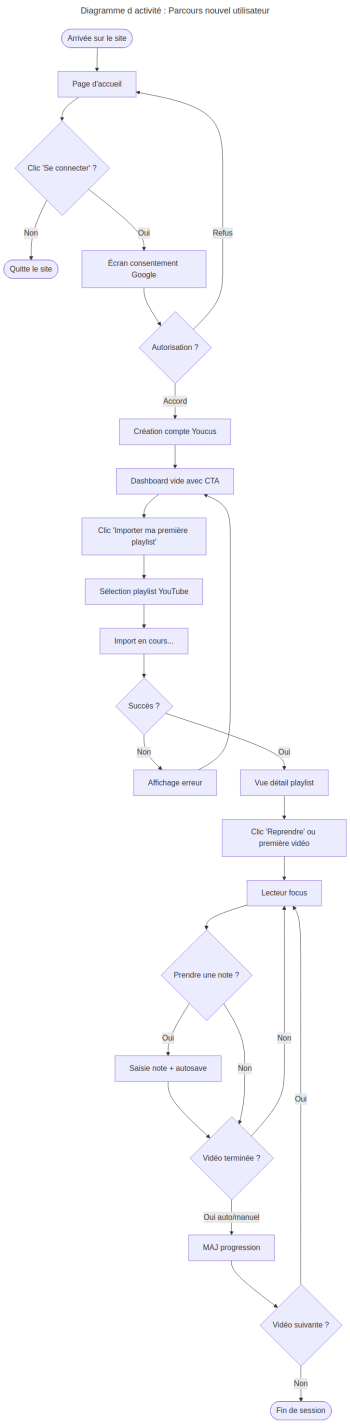


Figure 46. *Activité : parcours utilisateur*

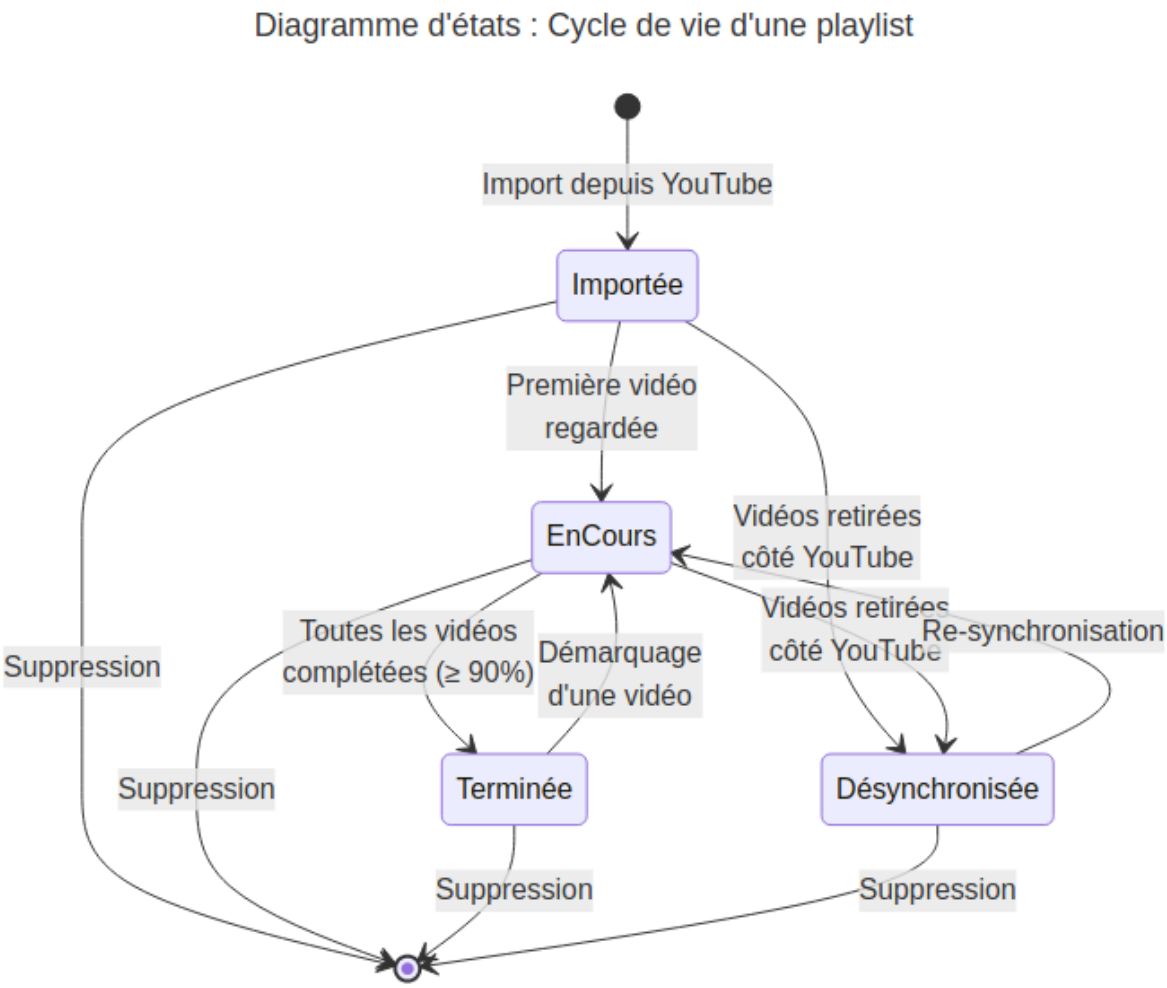


Figure 47. *États : cycle de vie d'une playlist*

Diagramme de déploiement Docker

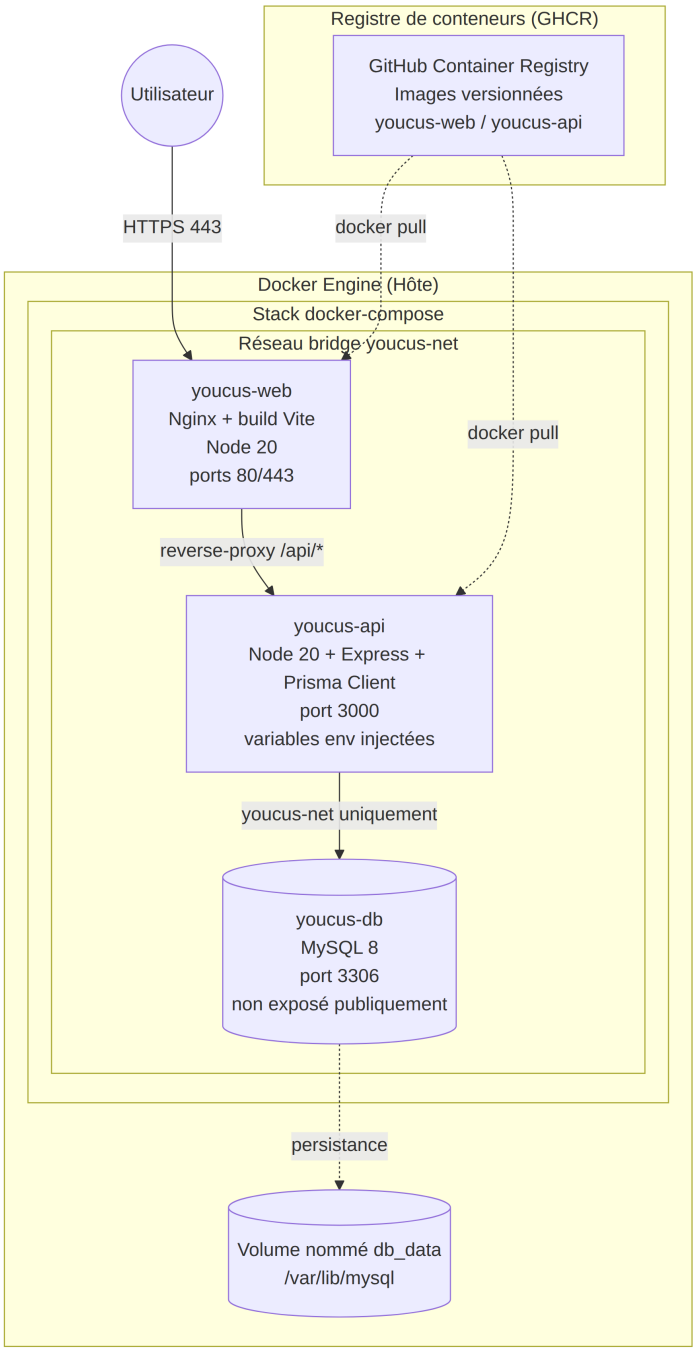


Figure 48. *Déploiement Docker*

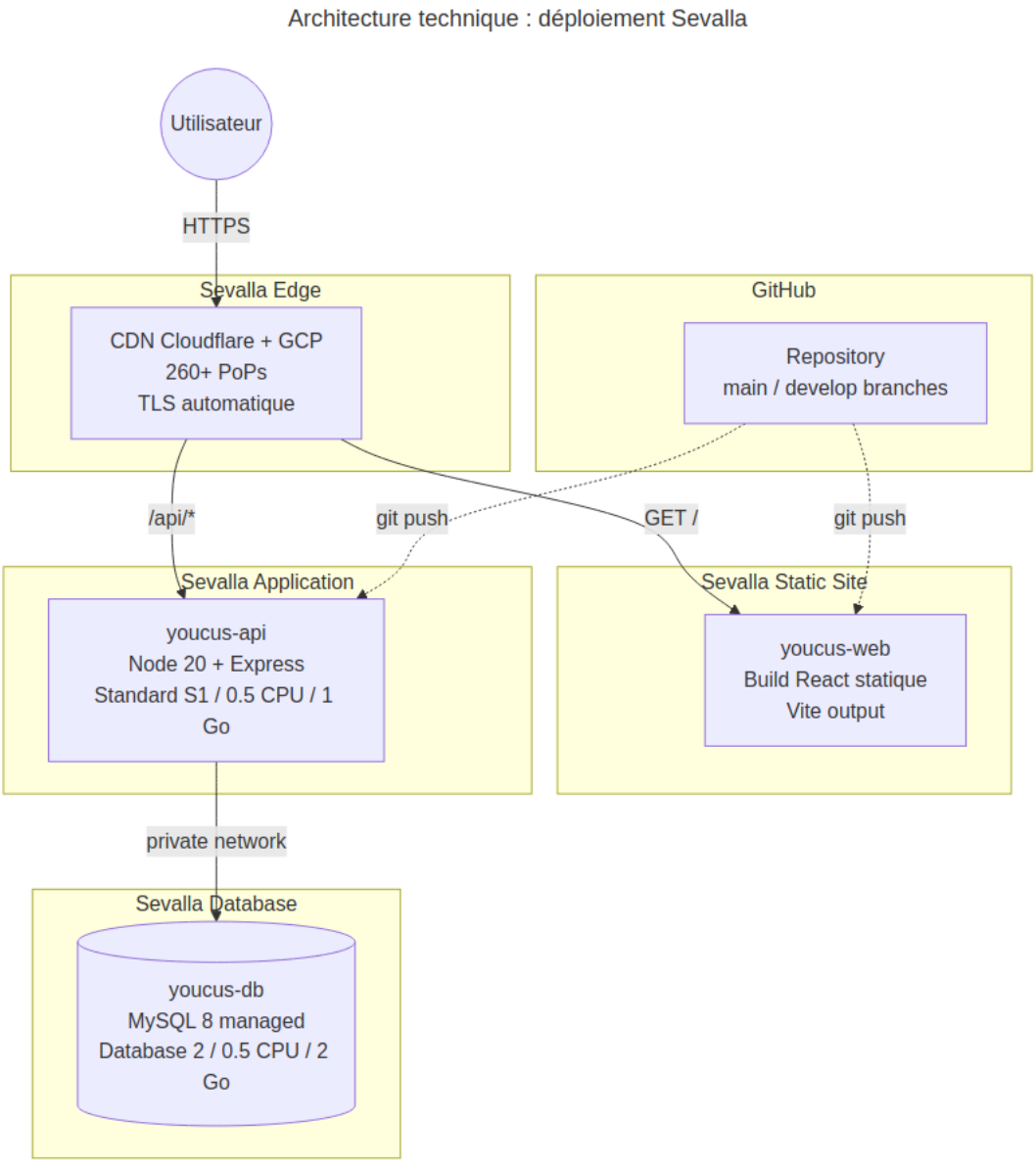


Figure 49. *Architecture Docker de l'environnement local*

Annexe D : User stories

Le noyau fonctionnel rédigé en amont du cadrage : six epics et vingt et une user stories, chacune avec sa priorité MoSCoW et ses critères d'acceptation. Le cadrage du 12 juin a réorganisé ce noyau en 8 epics Jira, puis le travail l'a enrichi de stories techniques (socle, CI/CD, interface) jusqu'aux 10 epics et 46 stories dont la section 4 rend compte. Certains critères ont par ailleurs évolué en cours de projet (chiffrement des jetons, SameSite, email de confirmation, durée des vidéos) : les choix finaux et leurs limites sont documentés aux section 19 et section 21. Ces stories sont donc à lire comme la spécification d'origine, pas comme un constat de l'état final.

Priorisation MoSCoW

- **MUST** : indispensable au MVP, sans quoi le produit n'a pas de sens.
- **SHOULD** : important, à livrer si possible avant la soutenance.
- **COULD** : agréable, à livrer si le temps le permet.
- **WON'T** : explicitement hors scope MVP, à mentionner comme évolution.

Format des stories

Chaque story suit le format standard :

> *En tant que [rôle], je veux [action], afin de [bénéfice].*

Suivi de critères d'acceptation testables sous forme de liste.

Epic 1 : Authentification et compte

US-001 : Connexion via Google (MUST)

En tant qu'utilisateur visiteur, je veux me connecter avec mon compte Google, afin d'accéder à mes playlists YouTube sans créer de nouveau compte.

Critères d'acceptation :

- Un bouton "Se connecter avec Google" est visible sur la page d'accueil.
- Le clic redirige vers l'écran de consentement Google.
- Le scope demandé inclut `youtube.readonly` et `userinfo.profile`.
- À la première connexion, un compte Youcus est créé automatiquement.
- Le refresh token Google est chiffré avant stockage en base.
- Une session est ouverte via cookie HttpOnly + SameSite=Lax + Secure.
- L'utilisateur est redirigé vers son dashboard.
- En cas d'erreur OAuth, un message d'erreur lisible est affiché.

US-002 : Déconnexion (MUST)

En tant qu'utilisateur connecté, je veux me déconnecter de Youcus, afin de sécuriser ma session sur un poste partagé.

Critères d'acceptation :

- Un bouton "Se déconnecter" est accessible depuis le menu utilisateur.
- Le cookie de session est immédiatement invalidé.
- L'utilisateur est redirigé vers la page d'accueil.

- L’access token YouTube reste valide côté Google (la session Youcus est dissociée).

US-003 : Suppression de compte (MUST)

En tant qu’utilisateur, je veux supprimer définitivement mon compte et toutes mes données, afin d’exercer mon droit à l’effacement (RGPD).

Critères d’acceptation :

- L’action est accessible depuis la page "Paramètres".
- Une double confirmation est demandée (modal + saisie du mot "SUPPRIMER").
- Toutes les données utilisateur sont supprimées (compte, playlists, notes, progression).
- Le refresh token Google est révoqué via l’endpoint Google.
- Un email de confirmation est envoyé à l’utilisateur.
- L’utilisateur est déconnecté et redirigé vers la page d’accueil.

US-004 : Export de ses données (SHOULD)

En tant qu’utilisateur, je veux exporter toutes mes données au format JSON, afin d’exercer mon droit à la portabilité (RGPD).

Critères d’acceptation :

- L’action est accessible depuis la page "Paramètres".
- Un fichier JSON est généré contenant : profil, playlists, vidéos associées, notes, progression.
- Le fichier est téléchargé via le navigateur.
- Aucun token n’est inclus dans l’export.

Epic 2 : Gestion des playlists

US-005 : Importer une playlist YouTube (MUST)

En tant qu’utilisateur connecté, je veux importer une de mes playlists YouTube, afin de la visionner dans Youcus.

Critères d’acceptation :

- Un bouton "Importer une playlist" est présent sur le dashboard.
- La liste de mes playlists YouTube est affichée (titre, miniature, nombre de vidéos).
- Je peux sélectionner une ou plusieurs playlists.
- À la confirmation, l’import démarre avec un indicateur de progression.
- Les métadonnées importées : titre, description, miniature, liste vidéos avec titre, durée, miniature, chaîne.
- Une notification confirme le succès ou affiche l’erreur.
- Une playlist déjà importée ne peut pas être ré-importée (proposition de "Synchroniser" à la place).

US-006 : Voir mes playlists (MUST)

En tant qu’utilisateur, je veux voir toutes les playlists que j’ai importées, afin de choisir laquelle reprendre.

Critères d’acceptation :

- Le dashboard liste les playlists avec : miniature, titre, nombre de vidéos, % progression.

- Les playlists sont triables par date d'import et par progression.
- Un champ de recherche filtre par titre.
- Un clic sur une playlist ouvre sa vue détail.
- Si aucune playlist : un état vide engageant avec un appel à l'action (CTA) "Importer ma première playlist".

US-007 : Voir le détail d'une playlist (MUST)

En tant qu'utilisateur, je veux voir le contenu d'une playlist, afin de choisir la vidéo à regarder.

Critères d'acceptation :

- La vue affiche titre, description, miniature, créateur de la playlist YouTube.
- Une barre de progression globale est visible en tête.
- La liste des vidéos s'affiche dans l'ordre original avec : miniature, titre, durée, statut (vue / non vue).
- Un clic sur une vidéo ouvre le lecteur focus.
- Un bouton "Reprendre" relance la première vidéo non vue.

US-008 : Synchroniser une playlist (SHOULD)

En tant qu'utilisateur, je veux synchroniser une playlist déjà importée, afin de récupérer les nouvelles vidéos ajoutées par le créateur.

Critères d'acceptation :

- Un bouton "Synchroniser" est présent dans la vue détail playlist.
- La synchro compare les vidéos en base avec celles de YouTube.
- Les nouvelles vidéos sont ajoutées en fin de liste.
- Les vidéos supprimées côté YouTube sont marquées "indisponible" mais conservées (avec leurs notes).
- Une notification résume les changements (X ajoutées, Y devenues indisponibles).

US-009 : Supprimer une playlist (MUST)

En tant qu'utilisateur, je veux retirer une playlist de Youcus, afin de désencombrer mon espace.

Critères d'acceptation :

- Un menu d'actions est accessible depuis la vue détail playlist.
- L'action "Supprimer" demande confirmation.
- La playlist, ses associations vidéos, les notes et progressions associées sont supprimées.
- L'utilisateur est redirigé vers son dashboard.

Epic 3 : Lecteur focus

US-010 : Lire une vidéo en mode focus (MUST)

En tant qu'utilisateur, je veux lire une vidéo dans un environnement sans distraction, afin de me concentrer sur le contenu.

Critères d'acceptation :

- Le lecteur utilise l'IFrame Player API de YouTube.
- Le paramètre `rel=0` désactive les suggestions de fin de vidéo.

- Aucun élément YouTube parasite n'est visible (logo cliquable masqué, etc.).
- L'affichage est centré, ratio 16:9 maintenu.
- Les contrôles standards sont disponibles : play/pause, vitesse, volume, fullscreen, qualité.
- La page de lecture ne contient que : lecteur, navigation playlist, zone notes.

US-011 : Naviguer entre les vidéos (MUST)

En tant qu'utilisateur, je veux passer à la vidéo suivante ou précédente, afin de suivre l'ordre pédagogique de la playlist.

Critères d'acceptation :

- Deux boutons "Précédent / Suivant" sont sous le lecteur.
- Une sidebar (ou drawer mobile) affiche la liste des vidéos cliquables.
- La vidéo courante est mise en évidence visuellement.
- Le changement de vidéo recharge le lecteur sans changer de page.
- L'URL change pour refléter la vidéo courante (deep linking).

Epic 4 : Progression

US-012 : Marquer une vidéo comme vue (MUST)

En tant qu'utilisateur, je veux marquer manuellement une vidéo comme terminée, afin de garder la trace de ma progression.

Critères d'acceptation :

- Un bouton "Marquer comme vue / non vue" est visible sous le lecteur.
- Le clic met à jour la progression instantanément (optimistic update).
- L'état est persisté en base.
- Une vidéo vue est marquée visuellement dans la liste (check, opacité, etc.).

US-013 : Marquage automatique de vidéo vue (SHOULD)

En tant qu'utilisateur, je veux que la vidéo soit marquée vue automatiquement lorsque je l'ai presque finie, afin de ne pas avoir à le faire manuellement.

Critères d'acceptation :

- Lorsque le watch progress dépasse 90 % de la durée, la vidéo est marquée vue.
- L'utilisateur peut désactiver ce comportement dans les paramètres (post-MVP).

US-014 : Voir la progression d'une playlist (MUST)

En tant qu'utilisateur, je veux voir où j'en suis dans une playlist, afin de savoir ce qu'il me reste.

Critères d'acceptation :

- Une barre de progression visible en tête de la vue playlist.
- Affichage "X / Y vidéos vues" et pourcentage.
- Temps restant estimé (somme des durées des vidéos non vues).

US-015 : Tableau de bord d'apprentissage (COULD)

En tant qu'utilisateur, je veux voir mes statistiques d'apprentissage, afin de mesurer mon engagement dans le temps.

Critères d’acceptation :

- Une page "Statistiques" est accessible depuis le menu.
- Affichage : temps cumulé cette semaine / ce mois / total.
- Nombre de vidéos vues cette semaine / total.
- Série de jours consécutifs d’utilisation.
- Un graphique en barres présente l’évolution des 8 dernières semaines.

Epic 5 : Notes

US-016 : Prendre une note texte sur une vidéo (MUST)

En tant qu’utilisateur, je veux rédiger une note attachée à une vidéo, afin de retenir ce que j’apprends.

Critères d’acceptation :

- Une zone de texte est visible à côté ou sous le lecteur.
- La saisie est sauvegardée automatiquement toutes les 5 secondes après modification.
- Un indicateur visuel signale l’état : "modifié", "sauvegarde...", "sauvegardé".
- La note est rechargée automatiquement à la prochaine visite de la vidéo.
- Le markdown basique est supporté (gras, italique, listes, liens).

US-017 : Modifier ou supprimer une note (MUST)

En tant qu’utilisateur, je veux éditer ou supprimer mes notes, afin de les maintenir à jour.

Critères d’acceptation :

- L’édition est inline directe dans la zone de notes.
- Un bouton "Vider la note" demande confirmation avant suppression complète.
- La suppression ramène la note à une chaîne vide (la ligne en base est conservée pour traçabilité).

US-018 : Notes timestampées (SHOULD)

En tant qu’utilisateur, je veux associer une note à un timestamp précis de la vidéo, afin de revenir exactement à ce moment plus tard.

Critères d’acceptation :

- Un bouton "Ajouter un repère ici" capture le timestamp courant du lecteur.
- Le repère apparaît dans la note sous forme `[12:34]` cliquable.
- Un clic sur le timestamp fait sauter le lecteur à ce moment.
- Des indicateurs visuels (points) signalent les timestamps sur la barre de progression du lecteur.
- Plusieurs repères peuvent être ajoutés à une même note.

US-021 : Prendre une note sur une playlist (MUST)

En tant qu’utilisateur, je veux rédiger une note attachée à une playlist entière (pas à une vidéo précise), afin de consigner mes objectifs d’apprentissage, mes conclusions globales ou mes prérequis perçus pour ce parcours.

Critères d’acceptation :

- Sur la vue détail d’une playlist, une zone « Note de playlist » est accessible (par exemple un panneau pliable en haut, ou un onglet « Notes »).

- La zone affiche une zone de texte multiligne avec support Markdown.
- La sauvegarde est automatique (debounce 5 secondes) avec indicateur visuel « Modifié » puis « Sauvegardé à HH:MM ».
- À l'ouverture suivante de la playlist, le contenu de la note est rechargé.
- Un utilisateur ne peut avoir qu'une seule note par playlist (contrainte UNIQUE sur `user_id`, `playlist_id`).
- La suppression de la playlist supprime aussi la note associée (cascade).
- La suppression du compte supprime toutes les notes playlist de l'utilisateur (cascade RGPD).
- Distinction visuelle claire entre note de playlist et notes de vidéos (icône différente, en-tête « Note de playlist » bien identifiée).

Epic 6 : Export des notes

US-019 : Exporter les notes d'une playlist en PDF (COULD)

En tant qu'utilisateur, je veux exporter en PDF les notes que j'ai prises sur une playlist, afin de garder une trace structurée hors ligne, la partager ou l'archiver.

Critères d'acceptation :

- Un bouton « Exporter les notes en PDF » est accessible depuis le menu utilisateur ou la page Paramètres.
- À l'ouverture, une liste affiche les noms et miniatures des playlists importées (avec nombre de vidéos et nombre de notes prises).
- Je peux sélectionner une seule playlist dans la liste.
- À la confirmation, l'export est généré pour la playlist sélectionnée.
- Le PDF contient : titre de la playlist en en-tête, puis pour chaque vidéo annotée : titre de la vidéo, durée, lien YouTube, contenu de la note.
- Les vidéos sans note sont omises par défaut (option pour les inclure avec mention « Aucune note »).
- Mise en page lisible (titres, pagination, en-tête, pied de page).
- Génération côté backend, téléchargement direct.

US-020 : Exporter les notes d'une playlist en Markdown (COULD)

En tant qu'utilisateur, je veux exporter en Markdown les notes que j'ai prises sur une playlist, afin de les réutiliser dans Obsidian, Notion ou un autre outil de prise de notes.

Critères d'acceptation :

- Un bouton « Exporter les notes en Markdown » est accessible depuis le menu utilisateur ou la page Paramètres.
- À l'ouverture, une liste affiche les noms et miniatures des playlists importées (avec nombre de vidéos et nombre de notes prises).
- Je peux sélectionner une seule playlist dans la liste.
- À la confirmation, l'export est généré pour la playlist sélectionnée.
- Un fichier `.md` est généré avec frontmatter YAML (titre playlist, date d'export, nombre de vidéos, nombre de notes).
- Structure : H1 titre de la playlist, H2 par vidéo annotée (avec lien YouTube et durée), contenu de la note dessous.
- Les vidéos sans note sont omises par défaut (option pour les inclure en commentaire).
- Téléchargement direct.

Récapitulatif par priorité

PRIORITÉ	NOMBRE	IDS
MUST	14	US-001, US-002, US-003, US-005, US-006, US-007, US-009, US-010, US-011, US-012, US-014, US-016, US-017, US-021
SHOULD	4	US-004, US-008, US-013, US-018
COULD	3	US-015, US-019, US-020
WON'T (MVP)	n/a	Téléchargement vidéo, IA, partage public, mobile natif, multi-utilisateurs

Traçabilité cahier des charges / user stories

Chaque fonctionnalité du cahier des charges (sections 2.1 et 2.2) est implémentée par une ou plusieurs user stories. Ce tableau permet au jury de vérifier la couverture exhaustive du périmètre fonctionnel et facilite la planification des tests d'acceptation.

FEATURE CAHIER	INTITULÉ	PRIORITÉ	USER STORIES	SPRINT
F1	Authentification Google OAuth 2.0	MUST	US-001	Sprint 1
F2	Déconnexion et suppression de compte RGPD	MUST	US-002, US-003	Sprint 1
F3	Import de playlists YouTube	MUST	US-005	Sprint 2
F4	Tableau de bord des playlists	MUST	US-006	Sprint 1-2
F5	Vue détail d'une playlist	MUST	US-007	Sprint 2
F6	Lecteur YouTube en mode focus	MUST	US-010, US-011	Sprint 3
F7	Marquage manuel d'une vidéo comme vue	MUST	US-012	Sprint 3
F8	Calcul de progression de playlist	MUST	US-014	Sprint 3
F9	Prise de notes texte par vidéo	MUST	US-016, US-017	Sprint 4
F10	Suppression d'une playlist	MUST	US-009	Sprint 4
F16	Prise d'une note sur une playlist	MUST	US-021	Sprint 4
F11	Notes timestampées	SHOULD	US-018	Sprint 5
F12	Synchronisation d'une playlist	SHOULD	US-008	Sprint 5
F13	Export PDF des notes d'une playlist	COULD	US-019	Sprint 5

FEATURE CAHIER	INTITULÉ	PRIORITÉ	USER STORIES	SPRINT
F14	Export Markdown des notes d'une playlist	COULD	US-020	Sprint 5
F15	Tableau de bord d'apprentissage	COULD	US-015	Sprint 5
Cahier, section 3.4	Droit à la portabilité RGPD (export JSON)	SHOULD	US-004	Sprint 5
(proposition US)	Marquage automatique de vidéo vue	SHOULD	US-013	Sprint 4

Lecture du tableau : les 16 fonctionnalités du cahier des charges (F1-F16) sont couvertes par 21 user stories, dont **11 fonctionnalités MUST (F1-F10, F16) déclinées en 14 user stories MUST** (certaines features couvrent plusieurs US). Les 2 user stories supplémentaires (US-004, US-013) sont des propositions issues du travail d'analyse fonctionnelle qui complètent la conformité RGPD (portabilité) et l'expérience utilisateur (marquage automatique). Aucune fonctionnalité du cahier n'est orpheline.

Découpage par sprint (suggestion)

> Vue indicative regroupant les user stories de ce document par sprint. Ce découpage en 6 sprints de 2 semaines était la projection d'origine. Le déroulé réellement suivi, resserré par le calendrier de l'alternance, est celui de la section 4.

SPRINT	SEMAINES	STORIES
Sprint 1 : Fondations	S1-S2	US-001, US-002, US-003, US-006 (état vide), schéma Prisma, Docker
Sprint 2 : Import	S3-S4	US-005, US-006 (liste), US-007
Sprint 3 : Lecture	S5-S6	US-010, US-011, US-012, US-014
Sprint 4 : Notes	S7-S8	US-016, US-017, US-021, US-013, US-009 + tests
Sprint 5 : Bonus	S9-S10	US-018, US-004, US-008, US-019, US-020 (selon temps)
Sprint 6 : Déploiement + Soutenance	S11-S12	Déploiement production, finalisation dossier + slides

Annexe E : Script de création de la base de données

Le script complet de l'état final du schéma, obtenu par `prisma migrate diff --from-empty --to-schema-datamodel prisma/schema --script`. La justification de chaque choix, types, contraintes d'unicité, index et comportement des cascades, est donnée à la section 10, dont cette annexe est le listing.

Table `User`

```
CREATE TABLE `User` (
    `id` VARCHAR(191) NOT NULL,
    `email` VARCHAR(191) NOT NULL,
    `displayName` VARCHAR(191) NOT NULL,
    `avatarUrl` VARCHAR(191) NULL,
    `googleId` VARCHAR(191) NOT NULL,
    `ytAccessToken` TEXT NULL,
    `ytRefreshToken` TEXT NULL,
    `ytTokenExpiry` DATETIME(3) NULL,
    `createdAt` DATETIME(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),
    `updatedAt` DATETIME(3) NOT NULL,

    UNIQUE INDEX `User_email_key`(`email`),
    UNIQUE INDEX `User_googleId_key`(`googleId`),
    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Table `Playlist`

```
CREATE TABLE `Playlist` (
    `id` VARCHAR(191) NOT NULL,
    `youtubeId` VARCHAR(191) NOT NULL,
    `title` VARCHAR(191) NOT NULL,
    `description` TEXT NULL,
    `thumbnailUrl` VARCHAR(191) NULL,
    `ownerId` VARCHAR(191) NOT NULL,
    `createdAt` DATETIME(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),
    `updatedAt` DATETIME(3) NOT NULL,

    INDEX `Playlist_ownerId_idx`(`ownerId`),
    UNIQUE INDEX `Playlist_ownerId_youtubeId_key`(`ownerId`, `youtubeId`),
    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Table `Video`

```
CREATE TABLE `Video` (
    `id` VARCHAR(191) NOT NULL,
    `youtubeId` VARCHAR(191) NOT NULL,
    `title` VARCHAR(191) NOT NULL,
    `durationSeconds` INTEGER NOT NULL DEFAULT 0,
    `thumbnailUrl` VARCHAR(191) NULL,

    UNIQUE INDEX `Video_youtubeId_key`(`youtubeId`),
    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Table `PlaylistVideo`

```
CREATE TABLE `PlaylistVideo` (
    `playlistId` VARCHAR(191) NOT NULL,
    `videoId` VARCHAR(191) NOT NULL,
    `position` INTEGER NOT NULL DEFAULT 0,

    INDEX `PlaylistVideo_videoId_idx`(`videoId`),
    PRIMARY KEY (`playlistId`, `videoId`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Table `Note`

```
CREATE TABLE `Note` (
    `id` VARCHAR(191) NOT NULL,
    `content` TEXT NOT NULL,
    `authorId` VARCHAR(191) NOT NULL,
    `videoId` VARCHAR(191) NULL,
    `playlistId` VARCHAR(191) NULL,
    `createdAt` DATETIME(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),
    `updatedAt` DATETIME(3) NOT NULL,

    INDEX `Note_authorId_idx`(`authorId`),
    UNIQUE INDEX `Note_authorId_videoId_key`(`authorId`, `videoId`),
    UNIQUE INDEX `Note_authorId_playlistId_key`(`authorId`, `playlistId`),
    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Table `Progress`

```
CREATE TABLE `Progress` (
    `id` VARCHAR(191) NOT NULL,
    `watchedSeconds` INTEGER NOT NULL DEFAULT 0,
    `completed` BOOLEAN NOT NULL DEFAULT false,
    `userId` VARCHAR(191) NOT NULL,
    `videoId` VARCHAR(191) NOT NULL,
    `updatedAt` DATETIME(3) NOT NULL,

    UNIQUE INDEX `Progress_userId_videoId_key`(`userId`, `videoId`),
    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Contraintes d’intégrité référentielle

```
ALTER TABLE `Note` ADD CONSTRAINT `Note_authorId_fkey` FOREIGN KEY
(`authorId`) REFERENCES `User`(`id`) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `Note` ADD CONSTRAINT `Note_videoId_fkey` FOREIGN KEY (`videoId`)
REFERENCES `Video`(`id`) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `Note` ADD CONSTRAINT `Note_playlistId_fkey` FOREIGN KEY
(`playlistId`) REFERENCES `Playlist`(`id`) ON DELETE CASCADE ON UPDATE
CASCADE;

ALTER TABLE `Playlist` ADD CONSTRAINT `Playlist_ownerId_fkey` FOREIGN KEY
(`ownerId`) REFERENCES `User`(`id`) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `PlaylistVideo` ADD CONSTRAINT `PlaylistVideo_playlistId_fkey`
FOREIGN KEY (`playlistId`) REFERENCES `Playlist`(`id`) ON DELETE CASCADE ON
UPDATE CASCADE;

ALTER TABLE `PlaylistVideo` ADD CONSTRAINT `PlaylistVideo_videoId_fkey`
```

```
FOREIGN KEY (`videoId`) REFERENCES `Video`(`id`) ON DELETE CASCADE ON UPDATE
CASCADE;

ALTER TABLE `Progress` ADD CONSTRAINT `Progress_userId_fkey` FOREIGN KEY
(`userId`) REFERENCES `User`(`id`) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `Progress` ADD CONSTRAINT `Progress_videoId_fkey` FOREIGN KEY
(`videoId`) REFERENCES `Video`(`id`) ON DELETE CASCADE ON UPDATE CASCADE;
```

Annexe F : Plan de tests détaillé

Recommandation du retour tuteur du 30 juillet. La section 20 expose la stratégie de test et les résultats agrégés ; cette annexe donne le détail suite par suite. Les nombres correspondent à l'état versionné du dépôt, celui que décrit la section 20 : cinquante-six cas côté serveur, vingt-cinq côté client, sur trente fichiers. Toutes les suites sont au vert sur la dernière exécution de l'intégration continue.

SUITE	TYPE	CE QU'ELLE VÉRIFIE	CAS	STATUT
`cache`	Intégration	Cache Redis en lecture-écriture différée, expiration, comportement dégradé quand Redis est absent	9	Vert
`note.service`	Unitaire	Création et mise à jour d'une note, contrôle du propriétaire, assainissement du Markdown	7	Vert
`youtube`	Intégration	Appels à l'API YouTube, traduction des erreurs, quota	6	Vert
`env.guard`	Unitaire	Refus de démarrer en production avec le secret de session par défaut, validation de l'adresse de rappel OAuth	6	Vert
`account.service`	Unitaire	Export RGPD, suppression de compte et effacement en cascade	4	Vert
`playlist.service`	Unitaire	Import d'une playlist, déduplication par identifiant YouTube	4	Vert
`youtubeAccount.service`	Unitaire	Liaison du compte YouTube, jetons et rafraîchissement	4	Vert
`migration`	Intégration	Migration de données du modèle 1:N vers N:N, sans perte	3	Vert
`playlist.merge`	Unitaire	Fusion de playlists, refus des cas interdits	3	Vert
`playlist.refresh`	Unitaire	Synchronisation depuis YouTube, refus sur playlist fusionnée	3	Vert
`progress.service`	Unitaire	Enregistrement de la progression, unicité par utilisateur et vidéo	2	Vert
`requireAuth`	Unitaire	Garde d'authentification, rejet sans session valide	2	Vert
`auth.service`, `health`, `logout`	Unitaire	Session, disponibilité du service, déconnexion	3	Vert

SUITE	TYPE	CE QU'ELLE VÉRIFIE	CAS	STATUT
Client (15 fichiers)	Composant	Rendu des écrans, états de chargement et d'erreur, interactions du lecteur et de l'éditeur	25	Vert

Les limites de ce plan sont assumées à la section 20 : absence de tests de bout en bout dans un navigateur, parcours OAuth non couvert de bout en bout, et absence de test de charge. Elles délimitent ce que ce tableau prouve et ce qu'il ne prouve pas.

Annexe G : Sources et références

Recommandation du retour tuteur. Les documentations officielles et les références consultées pendant le projet, regroupées ici pour que chaque choix technique puisse être remonté à sa source.

Interfaces de programmation externes. Documentation de l'API YouTube Data v3 (`developers.google.com/youtube/v3`), notamment les ressources `playlists` et `playlistItems` et le barème de quota qui justifie la stratégie de cache décrite à la section 17. Documentation de l'API du lecteur intégré (`developers.google.com/youtube/iframe_api_reference`), dont dépend le lecteur focus. Conditions d'utilisation des services d'API YouTube, qui fondent les contraintes énoncées à la section 6.

Authentification. Documentation OAuth 2.0 de Google (`developers.google.com/identity/protocols/oauth2`), pour le flux d'autorisation, les jetons de rafraîchissement et la configuration des adresses de redirection, au cœur de l'incident raconté à la section 14.

Sécurité. Documents de l'OWASP : le Top 10 des risques applicatifs, et les aide-mémoire consacrés à la protection contre les requêtes forgées et à la sécurité des sessions, qui fondent l'analyse de la section 19. Documentation de `helmet` pour les en-têtes de sécurité.

Persistance et cadre technique. Documentation de Prisma (`prisma.io/docs`), pour la modélisation, les migrations et le comportement des suppressions en cascade. Documentation de MySQL 8 pour les types, l'encodage `utf8mb4` et les limites d'indexation évoquées à la section 10. Documentation de Redis pour l'expiration des clés. Documentations d'Express, de React, de Vite, de TanStack Query, de Tailwind CSS et de Vitest.

Déploiement. Documentation de Sevalla pour la configuration des services, des variables d'environnement et des déploiements, et documentation de Docker et de GitHub Actions pour la chaîne d'intégration et de livraison décrite à la section 4.

Cadre de la certification. Référentiel du titre professionnel Concepteur Développeur d'Applications (RNCP niveau 6) et guide de rédaction du dossier de projet, qui fixent le plan suivi par ce document.

Annexe H : Business plan (extrait, document de cadrage non évaluant)

Ce business plan date du 7 juin 2026, il est antérieur à la première ligne de code. C'est un document de cadrage, versé ici comme pièce de contexte et non comme un élément à évaluer : il énonce ce que le projet visait, y compris des mesures de sécurité qui étaient alors des intentions. L'état réellement livré, et les limites que j'assume, sont décrits à la section 19 et à la section 21, qui font foi.

Il en est reproduit un extrait. Les chapitres purement commerciaux du document d'origine, analyse de marché, analyse concurrentielle, stratégie d'acquisition et indicateurs de pilotage, ont été retirés : ils relèvent d'une démarche entrepreneuriale qui n'est pas l'objet de ce dossier, et leurs chiffres de marché reposaient sur des sources de seconde main que je n'ai pas pu vérifier une à une. Ne subsistent que les chapitres auxquels le dossier renvoie réellement : le produit et son périmètre, le modèle économique dont les coûts d'hébergement sont aujourd'hui relevés sur facture, les risques, la feuille de route et la conclusion.

Le calendrier qu'il annonce appelle la même lecture. Les douze semaines prévues ici, en six sprints de deux semaines, étaient le plan normal du projet au 7 juin. Ce plan n'a pas tenu, et pour une raison extérieure au projet : le chantier d'infrastructure qui devait couvrir mes compétences de déploiement chez mon employeur a été écarté au profit d'autres priorités. Youcus est alors devenu le seul terrain possible pour ces compétences, dans une fenêtre déjà réduite. J'ai choisi d'accélérer plutôt que de réduire le périmètre. Le déroulé réel, avec ses dates, est raconté à la section 3 et à la section 4.

Un dernier point a été mis à jour depuis la rédaction, et c'est le seul : les coûts d'hébergement, au chapitre du modèle économique. Ils reposaient sur une estimation faite avant tout déploiement ; ils reposent maintenant sur la facture. Sur la période du 1er juillet au 1er août 2026, la production a coûté 3,90 $, répartis entre l'API (2,26 $) et la base de données (1,64 $), le front statique n'étant pas facturé. Rapporté au mois complet, cela représente environ 9 € d'hébergement. Deux remarques s'imposent : ce relevé porte sur une application en production mais sans charge d'utilisateurs réels, il constitue donc un plancher et non un coût en exploitation ; et il confirme au passage le choix d'architecture défendu à la section 7, servir le front en statique plutôt que par un processus, puisque c'est le seul des trois services à ne rien coûter.

1. Résumé exécutif

Youcus est une plateforme web d'apprentissage construite par-dessus YouTube. Elle s'adresse aux apprenants autodidactes (étudiants, développeurs en formation continue, personnes en reconversion) qui utilisent quotidiennement YouTube comme bibliothèque éducative mais souffrent des mécanismes d'engagement de la plateforme : recommandations algorithmiques, autoplay, shorts, commentaires. La promesse est simple : extraire la valeur pédagogique de YouTube sans son environnement de divertissement.

L'application repose exclusivement sur les API officielles de Google et YouTube. Elle permet d'importer ses playlists, de les visionner dans un lecteur épuré via le YouTube IFrame Player API, de suivre sa progression et de prendre des notes timestampées. L'architecture est conteneurisée avec Docker, organisée en couches contrôleurs / services / repositories (backend Express / TypeScript, frontend React / Vite, base de données MySQL). Aucun

contenu n’est dupliqué : Youcus est une couche d’usage par-dessus YouTube, dans le respect strict des CGU et de la conformité RGPD.

La singularité du projet réside dans la cohérence de ses choix : la promesse produit (protéger l’attention) est alignée avec un modèle économique cost-recovery non lucratif (autofinancement par un abonnement Premium symbolique à 2,99 € / mois, break-even atteint dès six abonnés) et avec une stratégie d’acquisition organique refusant la publicité payante. Le projet est conçu pour valider le Titre Professionnel Concepteur Développeur d’Applications et constitue une base réutilisable pour un service durablement autofinancé.

2. Présentation du produit

2.1 Proposition de valeur

> *"YouTube, sans la distraction. Vos playlists d’apprentissage, dans un environnement pensé pour la concentration."*

Pour qui : apprenants autodidactes consommant régulièrement du contenu éducatif sur YouTube. **Quel problème** : les mécanismes d’engagement de la plateforme (recommandations algorithmiques, shorts, commentaires, autoplay) dégradent la concentration et la qualité d’apprentissage. **Notre solution** : une couche d’apprentissage construite par-dessus YouTube, sans duplication de contenu, focalisée sur la concentration, la progression et la rétention. **Comment** : import de playlists via l’API officielle, lecture intégrée via le YouTube IFrame Player API, prise de notes timestampées, suivi de progression vidéo par vidéo.

2.2 Bénéfices utilisateur

- **Concentration préservée** : interface dépourvue de recommandations, shorts, commentaires et autoplay.
- **Progression mesurable** : marquage des vidéos vues et indicateur d’avancement playlist par playlist.
- **Notes contextualisées** : prise de notes timestampées, indissociables du contenu vidéo.
- **Reprise sans friction** : retour automatique à la dernière vidéo consultée et à son timecode exact.
- **Conformité légale** : aucune duplication de contenu, lecture via les API officielles, respect strict des CGU YouTube.

2.3 Différenciation produit

YouTube héberge une bibliothèque éducative gratuite d’une richesse exceptionnelle : cours d’universités prestigieuses (MIT OpenCourseWare, Stanford, Yale), bootcamps de développement (freeCodeCamp, Fireship, The Net Ninja), vulgarisation scientifique (3Blue1Brown, Veritasium, Kurzgesagt), apprentissage généraliste (Khan Academy, Crash Course). Cette richesse pédagogique reste pourtant largement sous-exploitée. Les mécanismes d’engagement de la plateforme (autoplay, recommandations algorithmiques, shorts, commentaires) captent l’attention vers le divertissement et fragmentent les sessions d’apprentissage. Le contenu éducatif coexiste avec un environnement conçu pour le contraire de l’apprentissage profond.

Youcus ne s’inscrit dans aucune catégorie existante de manière exacte. Les plateformes éducatives classiques (Coursera, edX, OpenClassrooms) hébergent leur propre contenu et

imposent leur propre rythme. Les LMS (Moodle, Canvas) sont conçus pour les institutions et les formateurs, pas pour l'apprenant individuel. Les outils de prise de notes (Notion, Obsidian) ne s'intègrent pas au support vidéo. Youcus se positionne dans l'angle mort : un environnement d'apprentissage individuel, centré sur le contenu YouTube que l'utilisateur a déjà identifié et structuré en playlists.

La force de ce positionnement tient à l'approche **« couche par-dessus YouTube »**. En ne dupliquant aucun contenu et en s'appuyant exclusivement sur les API officielles, Youcus évite les coûts d'hébergement vidéo, les questions de droits d'auteur et les risques de retrait de contenu (DMCA takedowns). Le rapport contenu / coût pour la plateforme est sans équivalent : l'intégralité du catalogue YouTube éducatif devient accessible sans dépense en licences. Pour l'utilisateur, c'est l'assurance que le service reste stable et accessible dans la durée, indépendamment des évolutions du marché des plateformes éducatives.

3. Modèle économique (perspective post-RNCP)

5.1 Philosophie tarifaire

Youcus n'est pas conçu comme un produit à maximisation de revenus. Le choix d'un modèle économique soutenable mais non lucratif découle directement de la philosophie du projet : Youcus revendique de protéger l'attention de ses utilisateurs ; il serait contradictoire d'optimiser à son tour la conversion ou l'engagement payant. L'objectif tarifaire est donc le seul équilibre des coûts d'exploitation : la version Premium doit suffire à payer l'hébergement, le domaine et les services techniques associés, sans dégager de marge significative. Cette posture renforce la cohérence du projet et constitue un argument de communication différenciant face aux services adoptant un modèle SaaS classique.

5.2 Coûts récurrents prévisionnels

Les coûts de fonctionnement de Youcus sont volontairement contenus grâce au choix d'une architecture éprouvée, d'hébergeurs européens à coût maîtrisé et d'une mutualisation des environnements de développement et de production.

Coûts en production (utilisateurs réels)

POSTE	COÛT MENSUEL	JUSTIFICATION
Hébergement Sevalla (API + base MariaDB ; front statique non facturé)	9 €	PaaS managée Kinsta, facturation horaire à l'usage réel, base de 1 Go, datacenter Frankfurt (RGPD)
Nom de domaine	1 €	~12 €/an amorti
Email transactionnel	0 €	Free tier Resend / Postmark (3 000 emails/mois)
Backups (S3-compatible)	1 à 2 €	Sauvegardes BDD quotidiennes
Sentry (monitoring erreurs)	0 €	Free tier 5 000 events/mois
API YouTube Data v3	0 €	Quota gratuit suffisant (10 000 unités/jour)
Tampon imprévus	5 €	Croissance, dépassements occasionnels
Sous-total production	**16 à 17 €**	

Coûts en développement et staging

L'environnement de développement engage des coûts marginaux pour un projet solo dev. Le dev tourne localement via Docker Compose sur la machine de l'auteur. Un environnement de staging est mutualisé sur l'infrastructure de production via un sous-domaine dédié, sans surcoût. La CI/CD est portée par GitHub Actions dans les limites de son tier gratuit.

POSTE	COÛT MENSUEL	JUSTIFICATION
Environnement local (Docker, IDE)	0 €	Machine personnelle, outils open source
Staging (mutualisé sur la prod via sous-domaine)	0 €	Pas d'infra dédiée requise au démarrage
GitHub repos privés + Actions	0 €	Free tier (2 000 min/mois) suffisant pour un solo
Outils dev (VS Code, Postman / Insomnia, DBeaver)	0 €	Tous gratuits ou open source
Sentry environnement dev	0 €	Même free tier que la prod, partagé
Sous-total développement	**0 €**	Mutualisation maximale

Total combiné (dev + staging + prod) : 16 à 17 €/mois. Le poste de coût reste exclusivement celui de la production. Cette mutualisation est viable tant que le projet reste solo dev ; à terme, si plusieurs développeurs collaborent ou si le staging devient critique pour des tests de charge, un environnement de staging dédié peut être provisionné (coût additionnel de ~5 €/mois pour un second environnement).

À ce stade, les coûts variables ne dépendent quasiment pas du nombre d'utilisateurs. Le seuil critique se situe non pas dans le compute mais dans l'éventuel ajout d'API IA (résumés automatiques) en option Premium ; ce coût serait alors mécaniquement couvert par les abonnements Premium associés.

5.3 Plans tarifaires

Compte tenu de l'objectif d'autofinancement, deux plans suffisent : un gratuit pour la majorité des utilisateurs, un Premium symbolique destiné à couvrir les frais.

PLAN	PRIX	PÉRIMÈTRE
Gratuit	0 €	Import jusqu'à 5 playlists, notes texte, progression, lecteur focus, export Markdown basique
Premium	2,99 € / mois ou 29 € / an	Playlists illimitées, notes timestampées, export PDF, statistiques d'apprentissage, fonctionnalités IA optionnelles

L'écart entre mensuel et annuel (~20 % de réduction sur l'annuel) reste modéré, en cohérence avec la posture non-commerciale : il ne s'agit pas d'optimiser la rétention par incentive financière, mais d'offrir une option de souplesse à ceux qui le souhaitent.

Une éventuelle offre **Educator** dédiée à des groupes d'apprenants (formateurs, enseignants, parents) reste à l'étude pour le long terme, sans engagement de calendrier.

5.4 Calcul du point mort

Avec un coût mensuel d'environ 16-17 € et un prix Premium à 2,99 € / mois, le service atteint son équilibre opérationnel à partir de :

- **6 abonnés mensuels** (6 × 2,99 € = 17,94 € de revenus mensuels)

Une fois ce seuil atteint, tout abonnement supplémentaire constitue une réserve pour couvrir les imprévus, financer les fonctionnalités IA optionnelles, et préparer une éventuelle migration d'infrastructure si le service grossit. Aucun revenu net (au sens de la maximisation des profits) n'est visé ; les surplus éventuels sont réinjectés dans le produit (amélioration de performance, ajout de fonctionnalités) ou conservés en réserve.

À titre indicatif, le seuil de viabilité long terme (incluant un dégagement de quelques heures de développement mensuel rémunérées) se situerait autour de 50 à 100 abonnés Premium, soit moins de 1 % d'une base utilisateurs de 10 000 inscrits gratuits.

5.5 Scénarios à 3 ans

Les projections ci-dessous illustrent la soutenabilité du modèle dans deux contextes contrastés. L'objectif n'est pas de viser une trajectoire de croissance startup, mais de vérifier que le service reste autofinancé quelle que soit la dynamique d'adoption.

Scénario réaliste

ANNÉE	INSCRITS CUMULÉS	ABONNÉS PREMIUM	REVENUS MENSUELS	COÛTS MENSUELS	SOLDE MENSUEL
An 1	300	6	18 €	17 €	+1 €
An 2	2 000	40	120 €	20 €	+100 € (réserve, R&D)
An 3	8 000	200	598 €	30 €	+568 € (réserve, IA)

Scénario pessimiste

ANNÉE	INSCRITS CUMULÉS	ABONNÉS PREMIUM	REVENUS MENSUELS	COÛTS MENSUELS	SOLDE MENSUEL
An 1	80	1	3 €	17 €	-14 € (autofinancé par l'auteur)
An 2	300	6	18 €	17 €	+1 €
An 3	1 000	20	60 €	18 €	+42 €

Dans le scénario pessimiste, le service reste viable : l'auteur du projet peut couvrir personnellement le déficit de la première année (~168 €), montant équivalent au coût annuel d'un abonnement à un service tiers comparable.

5.6 Scalabilité des coûts à grande échelle

Si le nombre d'utilisateurs croît significativement au-delà du scénario réaliste à 3 ans, plusieurs postes de coûts deviennent variables : hébergement, base de données, monitoring, email transactionnel. Le modèle reste néanmoins autofinancé, et le ratio revenus / coûts s'améliore à mesure que l'audience grossit.

PALIER	UTILISATEURS ACTIFS	COÛTS MENSUELS (DÉTAIL)	SEUIL BREAK-EVEN
MVP	< ; 1 000	~17 € (Sevalla facturé à l'usage + free tiers)	**6 abonnés × 2,99 € = 18 €/mois** (0,6 % de conversion)
Petite échelle	1 000 - 10 000	50-60 € (pods applicatif et base de taille supérieure, Sentry team, email tier 1)	**20 abonnés × 2,99 € = 60 €/mois** (0,2 % de conversion)
Échelle moyenne	10 000 - 50 000	130-150 € (cluster + BDD managed + backups étendus)	**50 abonnés × 2,99 € = 150 €/mois** (0,1 % de conversion)
Grande échelle	50 000 - 200 000	300-400 € (cluster HA + cache + CDN + monitoring)	**130 abonnés × 2,99 € = 389 €/mois** (< ; 0,1 % de conversion)
Forte croissance	> ; 200 000	1 000+ € (orchestration managée Kubernetes)	**350+ abonnés × 2,99 € = 1 047+ €/mois** (< ; 0,1 % de conversion)

Lecture du tableau : à chaque palier d'infrastructure, la colonne « Seuil break-even » indique combien d'utilisateurs doivent passer à l'abonnement Premium pour que le service s'autofinance. Par exemple, à grande échelle (50 000 à 200 000 utilisateurs), il suffit que 130 d'entre eux soient abonnés Premium (soit moins de 0,1 % de l'audience) pour générer les 389 € mensuels qui couvrent les ~390 € de coûts d'infrastructure du palier.

À chaque palier, le taux de conversion Premium nécessaire pour assurer l'autofinancement **diminue** à mesure que l'audience grossit : il passe de 0,6 % au MVP à moins de 0,1 % à grande échelle. La logique est mécanique : les coûts montent par paliers d'infrastructure (passage à des pods plus grands, puis d'une base simple à une base haute disponibilité), tandis que les revenus montent linéairement avec le nombre d'abonnés Premium.

Concrètement, plus le service grossit, plus son fonctionnement devient stable financièrement. Cela permet d'envisager sereinement les investissements de qualité de service (CDN, monitoring renforcé, support utilisateurs, éventuelles intégrations IA premium). À aucun palier réaliste le modèle n'exige une bascule vers une logique de maximisation des revenus.

Conclusion : Youcus est conçu pour pouvoir grandir sans changer de nature. Le passage du MVP à une plateforme servant 200 000 utilisateurs ne requiert aucune levée de fonds, aucun pivot de monétisation, ni aucun changement éthique sur le rapport à l'attention de l'utilisateur. C'est un modèle « scale-able sans devenir lucratif », rare dans le paysage SaaS actuel.

4. Risques et mitigations

Tout projet logiciel comporte des risques. Cette section identifie les principaux risques pesant sur Youcus et expose les mesures de mitigation prévues. Le format combine une analyse argumentée par catégorie et un tableau récapitulatif qui synthétise probabilité, impact et plan d'action pour chaque risque.

7.1 Risques techniques et produit

Changement des CGU YouTube ou modifications de l'API officielle. YouTube peut faire évoluer ses Conditions Générales d'Utilisation ou restreindre les fonctionnalités de l'API Data v3 et du player IFrame. Cette éventualité, peu fréquente dans l'histoire récente (les dernières modifications majeures datent de 2019 avec la suppression de l'API v2), reste néanmoins à surveiller. Mitigation : veille active sur les release notes officielles via l'abonnement à la newsletter Google Developers, conception modulaire isolant la couche d'accès YouTube derrière une interface unique (architecture en ports & ; adapters), ce qui permettrait un pivot vers une autre source si nécessaire (Vimeo, PeerTube, ou une intégration multi-sources).

Dépassement des quotas API YouTube. L'API Data v3 alloue 10 000 unités de quota par jour par projet. Chaque appel coûte un nombre variable d'unités (1 pour une lecture simple, 50 pour un search). Un usage intensif d'imports de playlists pourrait approcher cette limite. Mitigation : cache agressif côté serveur (les métadonnées d'une vidéo changent rarement après publication), pagination intelligente, traitement asynchrone via une file d'attente, demande de quota étendu auprès de Google si nécessaire (procédure documentée et fréquemment acceptée pour les services légitimes).

Évolutions du player IFrame YouTube. Le YouTube IFrame Player API peut subir des changements d'interface ou de comportement. Le risque est limité car cette API est utilisée par d'innombrables intégrations externes et Google maintient une compatibilité descendante. Mitigation : suivi des release notes et tests visuels automatisés couvrant les principaux scénarios de lecture (démarrage, pause, seek, fin de vidéo).

7.2 Risques marché et adoption

Concurrence directe par Google ou un acteur établi. Le risque le plus structurel est que Google ajoute nativement à YouTube un mode « apprentissage » ou « focus » intégrant les fonctionnalités de Youcus (suppression des distractions, prise de notes, suivi de progression). Mitigation : ce scénario est probable à long terme mais ne menace pas l'existence du projet à court ou moyen terme. La différenciation s'opérera sur la qualité produit, le positionnement éthique non-commercial (que Google ne peut pas adopter sans contredire son modèle économique) et la communauté construite autour du service. Une intégration partielle par YouTube serait plus probablement une validation du besoin qu'une élimination du marché.

Faible adoption initiale. Un produit de niche par construction (apprenants exigeants sur leur attention) prend du temps à trouver son public. Mitigation : MVP minimaliste permettant d'itérer rapidement sur les retours utilisateurs, niche claire et bien définie pour la communication initiale, acquisition organique patiente fondée sur le SEO long-tail et les communautés, modèle économique cost-recovery qui ne suppose pas une adoption massive pour rester viable.

7.3 Risques juridiques et sécurité

Conformité RGPD. Youcus collecte des données personnelles (identité Google, playlists importées, notes prises) et doit respecter le Règlement Général sur la Protection des Données. Mitigation : déclaration de traitement claire dans la politique de confidentialité, recueil du consentement explicite, chiffrement en base des tokens OAuth, droit à l'effacement implémenté (suppression de compte cascade sur l'ensemble des données associées), portabilité des données via export JSON, durée de conservation limitée pour les comptes inactifs, hébergement en Europe (Sevalla, centre de données de Francfort), qui

maintient les données dans l'Union. Cette localisation ne suffit pas à elle seule : Sevalla appartient au groupe Kinsta, société de droit américain, et l'analyse complète supposerait de vérifier les clauses contractuelles types encadrant un éventuel accès depuis les États-Unis. Je le signale comme une limite de ce document de cadrage, pas comme un point traité.

Sécurité de l'application. Comme toute application web manipulant des tokens OAuth et exposant des endpoints publics, Youcus est exposée aux risques classiques : vol de session, injection XSS ou SQL, attaques par force brute, vulnérabilités dans les dépendances. Mitigation : application stricte du référentiel OWASP Top 10 documenté dans le dossier sécurité, chiffrement AES-256-GCM des tokens OAuth en base, cookies de session HttpOnly + SameSite=Lax + Secure, headers de sécurité (Helmet), rate limiting sur les endpoints sensibles, scan automatisé des dépendances (Dependabot, npm audit en CI), validation Zod systématique des entrées.

7.4 Risques personnels

Contrainte de temps liée à l'alternance. L'auteur du projet est apprenti en alternance chez MBUSINESSEUROPE, où il développe la plateforme Néatemys. Le projet Youcus est mené en parallèle, sur le temps personnel disponible (soirées et week-ends), avec une fenêtre globale de 12 semaines (6 sprints de 2 semaines). Cette contrainte est le risque le plus tangible du projet. Mitigation : périmètre MVP volontairement restreint (11 fonctionnalités MUST, planning chiffré avec 20 % de marge), priorisation MoSCoW stricte permettant de descendre des features de MUST à SHOULD en cas de retard, MVP testable à mi-parcours (fin du sprint 3), plan B explicite de réduction de scope si le retard dépasse une semaine.

7.5 Tableau récapitulatif

RISQUE	CATÉGORIE	PROBABILITÉ	IMPACT	MITIGATION PRINCIPALE
Changement des CGU YouTube	Technique / Produit	Faible	Élevé	Veille, architecture modulaire
Dépassement quotas API	Technique / Produit	Moyenne	Moyen	Cache, pagination, demande de quota étendu
Évolutions player IFrame	Technique / Produit	Faible	Faible	Suivi release notes, tests visuels
Concurrence directe (Google)	Marché / Adoption	Faible (court terme)	Élevé	Différenciation produit et éthique
Faible adoption initiale	Marché / Adoption	Élevée	Moyen	MVP minimal, cost-recovery, niche claire
Non-conformité RGPD	Juridique / Sécurité	Faible	Élevé	Application complète du RGPD dès le MVP
Vulnérabilités sécurité	Juridique / Sécurité	Moyenne	Élevé	OWASP Top 10, scan dépendances, headers, rate limit
Contrainte de temps personnelle	Personnel	Élevée	Élevé	Périmètre restreint, MoSCoW strict, plan B scope

5. Roadmap produit

La roadmap de Youcus s'étend sur quatre horizons temporels distincts, chacun guidé par une logique de cohérence avec le modèle économique cost-recovery et la philosophie produit. Les fonctionnalités sont volontairement priorisées pour livrer rapidement un MVP utilisable puis enrichir le produit sans dépendre d'un investissement massif ni d'une croissance forcée.

8.1 Court terme : MVP (semaines 1 à 12)

Le MVP livré dans le cadre du projet RNCP couvre les fonctionnalités minimales suffisantes pour un usage réel : authentification OAuth Google, import de playlists YouTube via l'API officielle, lecteur de vidéo focus (sans recommandations, sans shorts, sans commentaires, sans autoplay), marquage des vidéos vues, indicateur de progression par playlist, prise de notes textuelles par vidéo. Cette base constitue le **produit utilisable** dès la fin du développement (sprint 6) et démontre la viabilité technique de l'approche « couche par-dessus YouTube ». Le périmètre est volontairement contraint pour permettre une livraison stable et testable dans la fenêtre temporelle imposée par l'alternance.

8.2 Moyen terme : enrichissement (mois 3 à 6 post-RNCP)

Une fois le MVP stabilisé et mis en ligne, les fonctionnalités SHOULD identifiées dans le cahier des charges peuvent être livrées : prise de notes timestampées (lien automatique entre la note et le timecode exact de la vidéo), export des notes au format PDF et Markdown, dashboard d'apprentissage personnel synthétisant la progression sur l'ensemble des playlists, statistiques de temps d'usage et d'engagement, mise en place du monitoring complet (Sentry production, logs structurés, alerting). Ces évolutions n'introduisent pas de nouveaux coûts d'infrastructure significatifs et restent compatibles avec le MVP Premium proposé à 2,99 € / mois.

8.3 Long terme : différenciation (mois 6 à 12)

À l'horizon d'un an, plusieurs fonctionnalités plus ambitieuses peuvent être intégrées si l'adoption se confirme. Les résumés IA optionnels (générés via un LLM tel que Claude ou Mistral, exclusivement sur déclenchement explicite de l'utilisateur) apportent une valeur ajoutée pédagogique tout en restant maîtrisables côté coût (le surcoût IA est intégralement couvert par les abonnements Premium concernés). La génération automatique de quiz à partir d'une vidéo permet à l'apprenant de tester sa rétention sans effort manuel. Le partage de playlists annotées via un lien public alimente la viralité douce évoquée dans la stratégie d'acquisition. Enfin, un mode **Educator** dédié aux groupes (formateurs, enseignants, parents structurant le visionnage de leurs enfants) peut être prototypé pour valider l'intérêt BtoB à coût symbolique.

8.4 Vision (an 2 et au-delà)

À horizon long, la promesse produit peut être étendue sans changer la nature du service. La transformation en Progressive Web App ou en application mobile native ouvre l'usage hors ligne et améliore l'expérience sur smartphone. L'intégration avec les outils tiers d'apprentissage de l'utilisateur (Notion, Obsidian, Anki, Readwise) permet de fluidifier la circulation des notes au-delà de Youcus, dans une logique d'interopérabilité et non d'enfermement. Une marketplace de playlists pédagogiques curées par des experts pourrait

constituer une couche de valeur communautaire, sans monétisation directe : la curation comme service public, financée par la même mécanique cost-recovery.

6. Conclusion

Youcus est une plateforme web d'apprentissage construite par-dessus YouTube, conçue pour permettre aux apprenants autodidactes de tirer pleinement parti d'un catalogue éducatif gratuit dont la richesse est aujourd'hui dégradée par les mécanismes d'engagement de la plateforme source. Sa proposition de valeur repose sur trois piliers : import et lecture focus de playlists YouTube via les API officielles, prise de notes timestampées, suivi de progression structuré. Aucun contenu n'est dupliqué, aucune dépendance n'est créée vis-à-vis d'un catalogue propre, aucun risque légal n'est pris sur les droits d'auteur.

La singularité du projet réside dans la cohérence systémique de ses choix. La promesse produit (protéger l'attention de l'utilisateur) est alignée avec son modèle économique (cost-recovery non lucratif, qui exclut toute incitation à la maximisation de l'engagement), avec sa stratégie d'acquisition (refus du growth marketing agressif et de la publicité payante), et avec son architecture technique (modulaire, conforme RGPD, hébergement européen). Cette cohérence n'est pas seulement une posture éthique : elle constitue un avantage structurel face à des acteurs établis dont le modèle économique impose à terme de capturer l'attention pour la monétiser.

L'objectif immédiat du projet est la validation du Titre Professionnel Concepteur Développeur d'Applications. Le développement du MVP en 12 semaines permettra de démontrer la maîtrise des compétences attendues sur l'ensemble du cycle de conception et de développement : analyse fonctionnelle, modélisation, architecture en couches, développement full-stack TypeScript, conteneurisation Docker, sécurisation, tests automatisés, déploiement et documentation. Au-delà du jury, Youcus constituera une base réutilisable pour un service durablement autofinancé, modeste mais utile à sa communauté d'apprenants. La trajectoire du projet dépendra moins de son potentiel commercial que du signal envoyé par ses premiers utilisateurs réels.

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE CHARGÉ
DE L'EMPLOI

DOSSIER PROFESSIONNEL (DP)

Nom de naissance - *RUKUNDO*
Nom d'usage - *RUKUNDO*
Prénom - *Ronaldo*
Adresse - *561 avenue d'Écully, 69410 Champagne-au-Mont-d'Or*

rukundoronaldo4@gmail.com

Titre professionnel visé

TITRE PROFESSIONNEL CONCEPTEUR DEVELOPPEUR D'APPLICATIONS

MODALITÉ D'ACCÈS :

☒ Parcours de formation
☐ Validation des Acquis de l'Expérience (VAE)

Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel. **Ce titre est délivré par le Ministère chargé de l'emploi.**

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente **obligatoirement à chaque session d'examen**.

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.

Il est consulté par le jury au moment de la session d'examen.

Pour prendre sa décision, le jury dispose :

1. des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.
2. du **Dossier Professionnel** (DP) dans lequel le candidat a consigné les preuves de sa pratique professionnelle.
3. des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation
4. de l'entretien final (dans le cadre de la session titre).

[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels du ministère chargé de l'Emploi]

Ce dossier comporte :

- pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;
- une déclaration sur l'honneur à compléter et à signer ;
- des documents illustrant la pratique professionnelle du candidat (facultatif)
- des annexes, si nécessaire.

Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.

http://travail-emploi.gouv.fr/titres-professionnels

Sommaire

Exemples de pratique professionnelle

PRATIQUE

PROFESSIONNELLE

DOSSIER PROFESSIONNEL (DP)

Activité-type 1 — Développer une application sécurisée

Exemple n°1 — *Refonte de l'interface de Néatemys : bibliothèque de composants django-cotton et migration de Bootstrap vers Tailwind CSS*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Le front de Néatemys est rendu par le serveur : des templates Django stylés avec Bootstrap, accumulés au fil des années, sans bibliothèque de composants commune. La migration vers Tailwind CSS était une proposition en attente depuis deux ans dans l'équipe ; c'est moi, l'alternant, qui l'ai menée.
En production, j'ai mené le chantier principal de mon année d'alternance : la construction d'une bibliothèque de composants d'interface avec django-cotton (211 templates de composants) et la migration progressive du CSS existant de Bootstrap vers Tailwind (37 templates repris). J'ai également réalisé les écrans des espaces métier (coaching, équipes, utilisateurs, offres, questionnaires, entreprises).
La méthode : je me suis d'abord documenté sur le développement par composants, je voulais des composants réutilisables, ce qui m'a conduit à django-cotton, puis j'ai configuré Tailwind en version 3, la plus éprouvée au moment du choix. J'ai découpé le chantier page par page, en analysant ce qui revenait plusieurs fois pour le transformer en composant.
Les deux vraies difficultés : le nettoyage de l'ancien code, on n'est jamais sûr qu'une classe n'est pas utilisée ailleurs, donc recherche systématique et vérification à la main ; et la conception des composants cotton eux-mêmes, choisir les bonnes variables, gérer les variantes conditionnelles. Le composant dont je suis le plus fier est l'un des premiers et des plus simples : le bouton retour, réutilisé sur de nombreuses pages, auquel j'ai ensuite appris la vraie navigation dans l'historique.

2. Précisez les moyens utilisés :

Django (templates), django-cotton, Tailwind CSS, Git et pull requests avec revues, Jira (83 tickets référencés dans mes commits sur la période), environnement de préproduction.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

Je contribue au même dépôt depuis août 2023 (environ 1 000 commits au total). La période décrite ici est celle de l'alternance iSCOD : 427 commits en 11 mois, à un rythme continu d'environ 39 commits par mois.

DOSSIER PROFESSIONNEL (DP)

Activité-type 1 — Développer une application sécurisée

Exemple n°2 ▸ *Transformer la page entreprise en hub de gestion autour d'un menu partagé*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

La plateforme lançait un nouveau service, le baromètre social, et c'est l'espace entreprise qui devait l'accueillir : c'est là que le responsable RH pilote ses services et, désormais, ses campagnes de mesure du climat social. Or la page entreprise accumulait des informations qui ne la concernaient pas, et chaque évolution la rendait moins lisible.
Je l'ai recentrée sur les informations propres à l'entreprise, puis réorganisée en hub de gestion autour d'un menu partagé, où chaque service de la plateforme trouve sa place.
Dans la foulée, j'ai construit la page de formulaire générique de création de campagnes avec son en-tête partagé, et promu ce formulaire générique dans la bibliothèque ui pour qu'il resserve ailleurs. J'ai aussi remplacé la marque codée en dur (deux images où le logotype était incrusté) par le logo venant du site, répété en pied de page. Le tout est passé par la revue d'équipe, dont j'ai intégré les retours.

2. Précisez les moyens utilisés :

Django (templates), django-cotton, Tailwind CSS, bibliothèque de composants ui, Git et revue de pull request, Jira, préproduction.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

DOSSIER PROFESSIONNEL (DP)

Activité-type 1 — Développer une application sécurisée

Exemple n°3 ▸ ***Construire le baromètre social : tableau de bord, éditeur de questionnaire et enquête papier***

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Néatemys édite une plateforme d'accompagnement RH en marque blanche : chaque entreprise cliente ouvre des services à ses salariés sous sa propre marque. Le baromètre social est le module qui lui permet de mesurer le climat social : des campagnes de questionnaires confidentielles auprès des salariés, en ligne ou par une enquête papier imprimable, puis une lecture agrégée des résultats.
J'ai construit le module de baromètre social. Le tableau de bord des campagnes affiche ses tuiles de comptage, et la page de campagne est découpée en un template par état (configuration, à venir, en cours, terminée) sous un en-tête partagé.
L'éditeur de questionnaire fonctionne par glisser-déposer : un questionnaire entier, un thème ou une question seule ; les éléments déjà sélectionnés se grisent, et la recherche dans le référentiel est extraite en module réutilisable. S'y ajoutent la génération du PDF d'enquête papier, et les statistiques de campagne groupées par thème, affichées une fois la campagne terminée.
Plusieurs briques sont passées par la préproduction et j'ai retravaillé le tableau de bord, le PDF et l'éditeur sur ces retours. C'est le chantier le plus récent de mon alternance : le tableau de bord employé a été fusionné le 31/07/2026.

2. Précisez les moyens utilisés :

Django (templates), django-cotton, Tailwind CSS, JavaScript (glisser-déposer), génération de PDF, Git et revues de pull request, Jira, préproduction.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

DOSSIER PROFESSIONNEL (DP)

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

Activité-type 1 — Développer une application sécurisée

Exemple n°4 ▸ ***Refondre l'espace médiation RH : pages collaborateur, composant nea_button et tarification en temps réel***

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

La médiation en entreprise est le cœur de métier de la plateforme : un collaborateur en difficulté ouvre un dossier depuis son espace, choisit le domaine d'expertise dont il a besoin et décrit sa situation, avant d'être pris en charge par la procédure adaptée. Cet écran d'entrée ne suivait plus la nouvelle identité visuelle du produit.

J'ai refondu l'espace médiation RH côté collaborateur : la page d'accueil de la médiation, puis le formulaire de choix de domaine, redessiné pour suivre la nouvelle maquette, et le formulaire de configuration.

À cette occasion j'ai créé le composant nea_button, que j'ai ensuite réutilisé sur la page de choix d'offre : un besoin né sur une page, transformé en composant de la bibliothèque.

J'ai aussi branché le service REST de tarification sur le configurateur solo, pour que le prix suive la configuration, et élargi la médiation à 10 participants.

2. Précisez les moyens utilisés :

Django (templates), django-cotton, Tailwind CSS, JavaScript, service REST de tarification, Git et revue de pull request, Jira, préproduction.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

DOSSIER PROFESSIONNEL (DP)

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

Activité-type 1 Développer une application sécurisée

Exemple n°5 ▸ Construire la page d'offre solo et les packs à options

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

La plateforme se vend par abonnements : l'entreprise choisit une offre, la configure selon le nombre de salariés à couvrir, et le prix se calcule en conséquence. La page d'offre est la vitrine commerciale du produit, et sa présentation est portée par les données de l'offre plutôt que codée dans les templates.
J'ai repris la présentation des offres. La page d'offre solo est redessinée autour d'un gabarit HTML porté par l'offre elle-même (offer_html), avec squelette de chargement, grande image et une bulle d'effectif qui suit le curseur quand l'utilisateur fait varier le nombre de salariés.
J'ai construit l'offre pack avec options, et la provision d'un abonnement par option complémentaire choisie. J'ai affiché les prix hors taxes sur la page de choix, la TVA restant visible dans le récapitulatif du bas.
J'ai enfin ajouté le bouton de renouvellement sur la page des abonnements, en réutilisant le flux d'extension existant plutôt que d'en créer un second.

2. Précisez les moyens utilisés :

Django (modèles et templates), JavaScript, django-cotton, Tailwind CSS, Git et revue de pull request, Jira.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

DOSSIER PROFESSIONNEL (DP)

Activité-type 1 — Développer une application sécurisée

Exemple n°6 ▸ ***Fiabiliser et sécuriser la page de paiement : échappement du HTML payeur et conditions au bon endroit***

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Le paiement en ligne clôt le parcours de souscription, et le payeur n'est pas toujours la personne qui a configuré l'offre : la plateforme lui adresse une page dédiée où il renseigne ses informations et règle. C'est l'écran le plus sensible du produit.

Sur le parcours de paiement, j'ai fait une passe de fiabilisation et de sécurité. J'ai échappé le HTML des formulaires payeur avec escapejs, pour qu'aucune donnée saisie ne puisse injecter de script dans la page.

J'ai retiré la page de conditions affichée au payeur, ses CGV étant déjà gérées sur la page de paiement elle-même : une correction de parcours autant que de droit.

J'ai corrigé une erreur de paiement, et mis en place la journalisation persistante des erreurs de la console JavaScript, pour pouvoir diagnostiquer ce que les utilisateurs rencontrent réellement.

2. Précisez les moyens utilisés :

Django (templates, filtres d'échappement), JavaScript, journalisation, Git et revue de pull request, Jira.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

DOSSIER PROFESSIONNEL (DP)

Activité-type 2

Concevoir et développer une application sécurisée organisée en couches

Exemple n° 1 ▸ *Un endpoint d'API pour le prix instantané sur la page de choix d'offre*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Les offres de la plateforme sont à prix variable : le tarif dépend du nombre de salariés que l'entreprise veut couvrir. Sur la page de choix d'offre, l'utilisateur fait varier ce nombre au curseur et le prix doit suivre en temps réel ; recharger toute la page à chaque variation aurait été lourd et lent.
J'ai conçu un endpoint d'API léger, dédié à ce seul calcul, que la page interroge dynamiquement. Je pense que c'était la bonne solution : il isole le calcul métier et ne touche à rien d'autre.
Sur la même page, j'ai aussi affiché le nombre d'utilisateurs, corrigé un cas de prix manquant, et remplacé la capture d'une RuntimeError par une vérification du palier maximal en amont : valider avant plutôt que rattraper après. Suite à la revue de ma PR, j'ai resserré les tests du prix instantané.

2. Précisez les moyens utilisés :

Django (vues d'API), JavaScript côté page, tests unitaires, Git et revue de pull request, Jira.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

DOSSIER PROFESSIONNEL (DP)

Activité-type 2

Concevoir et développer une application sécurisée organisée en couches

Exemple n° 2 ▸ *Signaler le renouvellement en attente de validation et encapsuler son calcul*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Les entreprises souscrivent des abonnements de séances qui se renouvellent. Quand un renouvellement attendait la validation de son règlement, l'interface ne l'indiquait pas clairement : l'abonné ne savait pas où il en était.
Une évolution courte mais représentative du travail en couches : j'ai ajouté ce signalement côté interface, puis simplifié le calcul du renouvellement en l'encapsulant dans la couche métier ; le template n'a plus qu'à afficher un état calculé au bon endroit.

2. Précisez les moyens utilisés :

Django (logique métier, templates), Git, Jira, environnement de préproduction.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

DOSSIER PROFESSIONNEL (DP)

Activité-type 2

Concevoir et développer une application sécurisée organisée en couches

Exemple n° 3 ▸ *Rétablir le temps réel de la salle d'attente visio après une montée de versions (redis-py 8, Channels 4)*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Les rendez-vous de la plateforme, médiations comme séances de coaching, se tiennent en visioconférence intégrée, avec une salle d'attente où chacun voit en temps réel qui est connecté. Ce temps réel repose sur un websocket porté par Django Channels et Redis.
Après la montée de redis-py en version 8 et de Channels en version 4, il ne fonctionnait plus. J'ai diagnostiqué la rupture de compatibilité entre les deux bibliothèques et rétabli le websocket en adaptant la configuration de la couche de canaux : le correctif final tenait en trois lignes de configuration, le travail réel, c'était le diagnostic.
Dans le même module, j'ai aussi différencié les couleurs des statuts de réservation pour les rendre lisibles d'un coup d'œil.

2. Précisez les moyens utilisés :

Django Channels, Redis, websockets, configuration Django, Git, Jira.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

Autre intervention du même esprit sur les données : limiter la vue entreprise aux données de la seule entreprise, un cloisonnement obtenu surtout en retirant du code qui n'avait pas sa place.

Activité-type 3

Préparer le déploiement d'une application sécurisée

Exemple n° 1 ▸ *Mettre en production dans la chaîne de validation de l'équipe : tests, préproduction, revue, documentation*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

L'application est en production chez des entreprises clientes : une régression atteint directement leurs utilisateurs. Chaque évolution que je livre suit donc la même chaîne de validation, et j'en connais chaque maillon.

D'abord je lis le code existant pour voir partout où ce que je modifie est appelé, puis je teste en local, manuellement et par les tests automatisés. Ensuite je déploie en préproduction. La développeuse de l'équipe relit ma pull request et teste de son côté ; elle passe alors le ticket à la directrice, qui teste à son tour. C'est seulement quand tout le monde a validé que le ticket part en production.

Je documente aussi ce que je livre : j'ai par exemple documenté la page de paiement dans l'espace de documentation partagé de l'équipe, à destination des futurs développeurs qui intégreront le projet et de la directrice. J'entretiens enfin les commandes de gestion des données qui accompagnent ces mises en production : j'ai fiabilisé migrate33, une commande de reprise de données, en la faisant échouer explicitement sur les utilisateurs sans profil plutôt que de la laisser continuer en silence, et renforcé la commande d'anonymisation des entreprises utilisée hors production.

2. Précisez les moyens utilisés :

Environnements local et préproduction, tests manuels et unitaires, pull requests et revues, espace de documentation partagé, Jira.

3. Avec qui avez-vous travaillé ?

En binôme avec Jola, la développeuse de l'équipe, qui relit et teste mes pull requests ; la directrice valide en dernier ; cycle préproduction → retours des utilisateurs → itération.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Néatemys*

Chantier, atelier, service ▸ *Équipe de développement*

Période d'exercice ▸ Du : *01/09/2025* au : *en cours (07/2026)*

5. Informations complémentaires *(facultatif)*

Titres, diplômes, CQP, attestations de formation

(facultatif)

Intitulé	Autorité ou organisme	Date
Certification professionnelle Développeur intégrateur web (niveau 5)	ifocop	20/06/2025
Certificat de scolarité Executive Bachelor Concepteur Développeur d'Applications (en cours, fin le 31/08/2026)	iSCOD	10/02/2026
Première année d'informatique (Bac+1)	Epitech	2021 - 2022

Documents illustrant la pratique professionnelle

(facultatif)

Intitulé

ANNEXES

(Si le RC le prévoit)