Colint School
Formation CDA 2025
Concepteur Développeur d'Applications
RNCP37873
54 Rue d'Autun, 71100 Chalon-sur-Saône

Diving O Club

Dossier de Projet CDA
Présentation et Défense

Informations du Projet

Candidat :	Lavier Kevin
Promotion :	3ème année Programmation informatique
Soutenance :	[Date]

Ce dossier présente le projet réalisé dans le cadre de la formation
Concepteur Développeur d'Applications (CDA) de Colint School

Année académique 2025-2026

Table des matières

1 Présentation personnelle et du projet **5**
- 1.1 Rôle du candidat et contexte . . . 5
- 1.2 Problématique et objectifs SMART . . . 6
- 1.3 Liens utiles . . . 9

2 Cadrage et cahier des charges **11**
- 2.1 Objectifs métier, techniques et pédagogiques . . . 11
- 2.2 Cibles et parties prenantes . . . 12
- 2.3 Exigences fonctionnelles . . . 16
  - 2.3.1 Fonctionnalités « Front Office » . . . 16
  - 2.3.2 Fonctionnalités « Back Office » . . . 16
  - 2.3.3 Découpage Front / Back (fonctionnalités clés) . . . 19
  - 2.3.4 L'utilisateur (public) . . . 19
  - 2.3.5 Confidentialité . . . 20
  - 2.3.6 Droits d'accès . . . 20
  - 2.3.7 Authentification . . . 20
- 2.4 Exigences et choix techniques . . . 21
  - 2.4.1 Exigences . . . 21
  - 2.4.2 Choix . . . 22
- 2.5 Définition du MVP . . . 26
- 2.6 Roadmap . . . 35
- 2.7 Liens utiles . . . 35

3 Méthodologie et organisation **37**
- 3.1 Gestion de projet avec GitHub . . . 37
- 3.2 Rituels de suivi du projet . . . 37
  - 3.2.1 User Stories et estimation de temps . . . 38
- 3.3 Versioning GitHub et conventions . . . 39
- 3.4 Planification et outils de suivi . . . 41
- 3.5 Estimation de temps et planification . . . 41
- 3.6 Liens utiles . . . 42

4 Conception fonctionnelle et technique **45**
- 4.1 Use Cases et diagrammes UML . . . 45
- 4.2 Diagrammes de séquence . . . 47
- 4.3 Conception de l'interface graphique . . . 49
  - 4.3.1 Zoning . . . 49
  - 4.3.2 Wireframes . . . 50
  - 4.3.3 Maquettage . . . 51
  - 4.3.4 Outils de conception et diagrammes . . . 52
  - 4.3.5 Charte graphique . . . 52
- 4.4 Conception de base de données . . . 52

4.4.1 MCD (Modèle Conceptuel de Données) . . . 53
4.4.2 MLD (Modèle Logique de Données) . . . 55
4.4.3 MPD (Modèle Physique de Données) . . . 57
4.5 Architecture 3 tiers . . . 61
4.6 Liens utiles . . . 62
5 Architecture 3 tiers 63
5.1 Architecture 3 tiers . . . 63
5.1.1 Couche Présentation (Frontend) . . . 63
5.1.2 Couche Logique Métier (Backend) . . . 66
5.1.3 Couche Données (Database) . . . 69
5.1.4 Communication entre les tiers . . . 70
5.1.5 Avantages de l'architecture 3 tiers . . . 71
5.2 Développement Frontend . . . 72
5.3 Développement Backend . . . 74
5.4 Gestion des données . . . 76
5.5 Liens utiles . . . 78
6 Sécurité applicative et RGPD 79
7 Tests et qualité logicielle 83
7.1 Stratégie de tests . . . 83
7.1.1 Tests unitaires . . . 83
7.1.2 Tests d'intégration . . . 85
7.1.3 Tests End-to-End (E2E) . . . 87
7.2 Tests de performance . . . 88
7.2.1 Scénarios de performance testés . . . 88
7.2.2 Exemple de script de test de performance . . . 89
7.2.3 Interprétation des résultats . . . 89
7.2.4 Analyse des performances front-end avec Lighthouse . . . 89
7.2.5 Tests d'accessibilité avec WAVE . . . 90
7.3 Qualité du code avec SonarQube . . . 90
7.3.1 Objectifs de l'analyse qualité . . . 90
7.3.2 Métriques surveillées . . . 91
7.3.3 Intégration dans la CI/CD . . . 91
7.3.4 Exemple de pipeline CI GitHub . . . 91
7.3.5 Conclusion . . . 92
7.4 Liens utiles . . . 92
8 Déploiement et CI/CD 93
8.1 Containerisation avec Docker . . . 93
8.2 Pipeline CI/CD avec GitHub Actions . . . 95
8.3 Documentation et monitoring . . . 100
8.4 Liens utiles . . . 104
9 Veille technologique et sécurité 105
9.1 Veille technologique stack . . . 105
9.2 Bonnes pratiques sécurité . . . 106
9.3 Application au projet . . . 107
9.4 Liens utiles . . . 109
10 Bilan et retour d'expérience (REX) 111
10.1 Objectifs atteints et non atteints . . . 111
10.2 Difficultés rencontrées et solutions . . . 111

10.3 Dettes techniques et apprentissages 113
10.4 Liens utiles 114
11 Conclusion et remerciements **115**
11.1 Synthèse du projet 115
11.2 Perspectives d'évolution 116
11.3 Remerciements 117
11.4 Déploiement et documentation 117
11.4.1 Docker 118
11.4.2 GitHub (code source) 119
11.4.3 CI/CD 120
11.4.4 SonarQube 120
11.4.5 Swagger 121
11.5 Liens utiles 123

Chapitre 1
Présentation personnelle et du projet

1.1 Rôle du candidat et contexte

Mon rôle : En tant que concepteur-développeur full-stack, mon rôle consiste à identifier et analyser les besoins afin d'apporter des solutions numériques concrètes et adaptées aux problématiques rencontrées. Mes expériences précédentes, à la fois en formation et en milieu professionnel, m'ont permis d'acquérir un éventail de compétences suffisamment large pour intervenir efficacement sur l'ensemble du cycle de développement. Je suis en autonomie sur l'architecture, la conception, les choix techniques et le développement, tout en tenant compte des retours des parties prenantes du club test "Aquaclub 21".

Contexte organisationnel : Je suis membre actif d'un club associatif de plongée affilié à la FFESSM depuis quatre ans et j'ai intégré le comité directeur. J'interviens sur la communication, la gestion du site internet et la coordination numérique du club. Comme dans de nombreuses structures associatives, l'organisation repose essentiellement sur le bénévolat, avec des ressources humaines et techniques limitées. Cela implique une forte autonomie dans la gestion des outils numériques pour les membres du comité.

Aujourd'hui, la gestion du club s'appuie sur une combinaison d'outils dispersés : VPDive, Google Drive, formulaires externes, tableurs partagés, messageries, documents papier. Cette fragmentation entraîne des manipulations manuelles répétitives, une perte d'informations, un risque d'erreur élevé et une charge administrative importante. Les retours du comité montrent une perte de 1 à 3 heures par semaine pour les bénévoles chargés du suivi des adhésions, certificats médicaux, inscriptions et paiements.

Un autre élément clé du contexte métier est le niveau d'aisance numérique hétérogène des pratiquants. Dans le club test Aquaclub 21, 64,6% ont plus de 46 ans, dont 30% ont plus de 61 ans. Ils peuvent facilement se retrouver en difficulté face à des interfaces complexes ou à des processus éclatés. Ils privilégient la simplicité, la lisibilité et l'automatisation.

Cette réalité accroît la nécessité d'un outil centralisé, clair et accessible, permettant de réduire les frictions, d'éviter les erreurs et de rendre les démarches (adhésion, certificats, inscriptions) plus intuitives. En tant que membre du comité, je suis régulièrement confronté à ces difficultés, ce qui m'a permis d'identifier précisément les points de friction et les besoins réels des utilisateurs.

Face à ces contraintes, les échanges avec le comité ont permis d'identifier le besoin d'une plateforme unifiée, robuste et adaptée aux processus du club associatif. L'objectif est de :

- centraliser les informations
- améliorer l'expérience des membres, y compris les moins à l'aise avec le numérique
- simplifier les démarches administratives
- automatiser les tâches récurrentes
- renforcer la fluidité des processus internes en développant des fonctionnalités adaptées

Le besoin métier principal consiste donc à mettre en place un outil simple, cohérent et efficace, capable d'accompagner la gestion quotidienne du club dans un environnement associatif exigeant mais limité en ressources.

Durée et planning : J'ai démarré mon alternance le 1er septembre 2025, à raison de trois jours en entreprise (lundi, mardi, mercredi) et deux jours en formation (jeudi et vendredi). Ce

dossier a été rédigé durant la phase de conception du projet, de septembre à décembre 2025.

Planning du projet *Diving O Club* :

Période	Objectifs clés
25 sept. — 20 déc. 2025	**Conception et cadrage** : deux phases d'idéation, analyse des besoins, modèles UML, MCD/MLD, wireframes, maquettes, backlog, choix techniques.
8 janv. — 17 janv. 2026	**POC** : validation de l'architecture (Next.js / Nest.js / PostgreSQL), mise en place de l'environnement Docker, affichage d'un club test avec membres et événements seedés.
22 janv. — 14 fév. 2026	**MVP** : création de compte, connexion, recherche du club test, consultation des membres et événements.
19 fév. — 25 avril 2026	**Version Bêta** : profils utilisateurs, certificats médicaux, calendrier, inscriptions, rôles (admin/moniteur/adhérent), gestion des membres et événements.
30 avril — 6 juin 2026	**Version 1** : multi-clubs, création et validation de club, invitations par email, demandes d'adhésion, paiements HelloAsso, tableaux de bord.

QQOQCP :

- **Quoi :** Application web unifiée permettant de gérer un club de plongée associatif : adhérents, certificats médicaux, événements, inscriptions, rôles, paiements et administration.
- **Qui :** Clubs FFESSM et associations subaquatiques reposant majoritairement sur le bénévolat, avec un public adulte et une aisance numérique très hétérogène.
- **Où :** Application web mobile-first, accessible également sur ordinateur et tablette, déployée d'abord dans un club pilote (Aquaclub21).
- **Quand :** Projet développé du 25 septembre 2025 au 6 juin 2026 (POC → MVP → Bêta → V1).
- **Comment :** Architecture 3-tiers *(Next.js - Nest.js - PostgreSQL + MongoDB)* avec intégration HelloAsso en V1 et pilotage via GitHub Projects.
- **Pourquoi :** Remplacer la fragmentation des outils actuels par une solution unique, simple et accessible, afin de réduire la charge administrative et d'améliorer l'expérience des adhérents.

Pitch QQOQCP : Je développe une application web mobile-first destinée aux clubs associatifs de plongée pour centraliser la gestion des adhérents, des certificats médicaux, des événements, des inscriptions, des rôles et des paiements.

L'application repose sur une architecture 3-tiers *(Next.js - Nest.js - PostgreSQL + MongoDB)* et sera testée en conditions réelles dans un club pilote (Aquaclub21) avant une version finale début juin 2026.

L'objectif est de remplacer la multiplicité des outils actuels par une solution unique, fiable et simple d'usage, afin de réduire la charge administrative des bénévoles et d'améliorer l'expérience des adhérents lors de l'utilisation de l'application.

1.2 Problématique et objectifs SMART

Problématique centrale :

Comment accompagner efficacement un public majoritairement adulte et peu technophile, tout

en réduisant significativement la charge administrative des clubs, dans un contexte où les outils actuels sont fragmentés, peu ergonomiques et insuffisamment adaptés aux processus associatifs réels ?

La gestion quotidienne d'un club associatif de plongée repose aujourd'hui sur une multitude d'outils hétérogènes : VPDive, tableurs, formulaires externes, Google Drive, messageries, documents papier. Cette fragmentation génère :

— une navigation complexe pour un public majoritairement adulte et peu technophile.
— des manipulations manuelles répétitives,
— une perte ou duplication d'informations,
— un risque d'erreur élevé,
— une surcharge administrative estimée entre 1 et 3 heures par semaine pour les bénévoles du comité,

Au sein du club pilote (Aquaclub21), 64,6 % des adhérents ont plus de 46 ans dont 27,1 % plus de 61 ans, ce qui renforce le besoin d'une interface simple, lisible et guidée.

Ces difficultés impactent :

— l'expérience utilisateur,
— la satisfaction globale des adhérents,
— la qualité du suivi administratif (certificats, paiements, inscriptions),
— la sécurité (certificats non valides),
— la participation aux activités (mauvaises informations, pertes d'inscriptions),

En tant que membre du comité, je suis confronté directement à ces problèmes, ce qui m'a permis d'identifier avec précision les points de friction, leurs impacts et les besoins concrets du terrain.

Objectifs SMART

Les objectifs du projet ont été formalisés selon la méthode SMART (Spécifique, Mesurable, Atteignable, Réaliste et Temporellement défini) afin de garantir un pilotage précis et vérifiable.

Indicateurs de succès quantifiés (KPIs)

Afin d'évaluer objectivement la réussite globale du projet, trois indicateurs clés (KPIs) ont été définis, chacun associé à une valeur cible et à un mode de mesure :

— **Taux d'adoption** : au moins **20 utilisateurs actifs** durant la phase bêta. *Mesure : analyse des logs de connexion et du nombre de comptes actifs.*
— **Gain de temps et réduction des relances** : diminuer d'au moins **50%** le volume de relances manuelles sous 2 mois. *Mesure : comptage hebdomadaire des relances manuelles réalisées par le comité.*
— **Satisfaction utilisateur** : atteindre **90% de satisfaction** en fin de phase bêta. *Mesure : questionnaire anonyme envoyé aux adhérents du club pilote.*

Objectif SMART 1 — Accessibilité des adhérents non technophiles
Indicateur de réussite : 80% des adhérents complètent l'inscription sans assistance, mesuré via le taux de complétion et le nombre de demandes d'aide (objectif $<$ 10 demandes).

— **Spécifique :** Simplifier l'usage de la plateforme pour les adhérents peu à l'aise avec le numérique.
— **Mesurable :** Atteindre **80% de parcours complets** réalisés sans assistance, mesurés via le taux de complétion et le nombre de demandes d'aide.

— **Atteignable :** Parcours guidé, interface mobile-first, formulaires simplifiés, notifications claires.
— **Réaliste :** Besoin confirmé par l'analyse utilisateurs du club (moyenne d'âge 48+ ans).
— **Temporel :** Mesuré pendant la phase bêta (**février** $\rightarrow$ **avril 2026**).

Objectif SMART 2 — Réduction de la charge administrative
Indicateur de réussite : Temps de gestion réduit à moins de 60 minutes/semaine, mesuré via un suivi hebdomadaire du temps passé par les bénévoles.
— **Spécifique :** Réduire le temps consacré aux relances et au suivi manuel (certificats, inscriptions).
— **Mesurable :** Réduire le temps administratif de **1–3 h/semaine à 30–60 min/semaine**, mesuré via le suivi du temps passé et les relances automatiques vs manuelles.
— **Atteignable :** Automatisation des relances, centralisation des données, flux simplifiés.
— **Réaliste :** Charge actuelle estimée à 1–3 h/semaine par bénévole.
— **Temporel :** Mesure effectuée entre **février** $\rightarrow$ **avril 2026**.

Objectif SMART 3 — Adoption de la Bêta par le club pilote
Indicateur de réussite : Au moins 20 utilisateurs actifs et 20 inscriptions à des événements dans le mois suivant le déploiement de la bêta.
— **Spécifique :** Déployer une version fonctionnelle couvrant profil, certificats, événements et inscriptions.
— **Mesurable :** Atteindre $\geq$ **20 utilisateurs actifs** et $\geq$ **20 inscriptions**, mesurés via logs de connexion et statistiques.
— **Atteignable :** Périmètre MVP limité et validé.
— **Réaliste :** Technologies maîtrisées (Next.js / Nest.js / PostgreSQL).
— **Temporel :** Objectif mesuré entre **février** $\rightarrow$ **avril 2026**.

Objectif SMART 4 — Performance et stabilité de l'API
Indicateur de réussite : 95% des requêtes serveur sous 300 ms, mesuré via l'outil de monitoring en production.
— **Spécifique :** Garantir une API performante et stable pour une expérience fluide.
— **Mesurable :** Assurer que **95% des requêtes** répondent en moins de **300 ms**.
— **Atteignable :** Optimisation des endpoints, mise en cache, bonnes pratiques Nest.js.
— **Réaliste :** Aligné avec les standards des architectures 3-tiers.
— **Temporel :** Mesuré en continu à partir de la **Bêta (février 2026)**.

Objectif SMART 5 — Sécurisation des données et conformité RGPD
Indicateur de réussite : 100% des routes sensibles protégées et aucun incident de sécurité durant la phase bêta.
— **Spécifique :** Protéger les données personnelles des adhérents.
— **Mesurable :** Vérifier que **100% des routes sensibles** sont protégées (JWT + RBAC), via audit interne.
— **Atteignable :** Cookies HttpOnly, RBAC, validation stricte des schémas.
— **Réaliste :** Aligné avec les exigences légales du RGPD.
— **Temporel :** Implémenté pour la **Bêta Release (avril 2026)**.

Bénéfices attendus : L'application a pour objectif de centraliser l'ensemble des opérations nécessaires à la gestion d'un club de plongée dans un seul outil, afin de simplifier le travail des encadrants et d'améliorer l'expérience des adhérents. Elle remplace plusieurs outils dispersés

par une plateforme unique, intuitive et maintenable dans le temps. La gestion des adhérents, des événements, des certificats et des inscriptions est unifiée, ce qui réduit les risques d'erreur et de perte d'information. Grâce à une interface adaptée aux publics peu familiers du numérique, l'application fluidifie la participation aux activités et diminue les frictions administratives. Enfin, la solution pourra évoluer à l'échelle fédérale afin d'assurer une cohérence numérique au sein des structures FFESSM.

Diagramme de contexte :

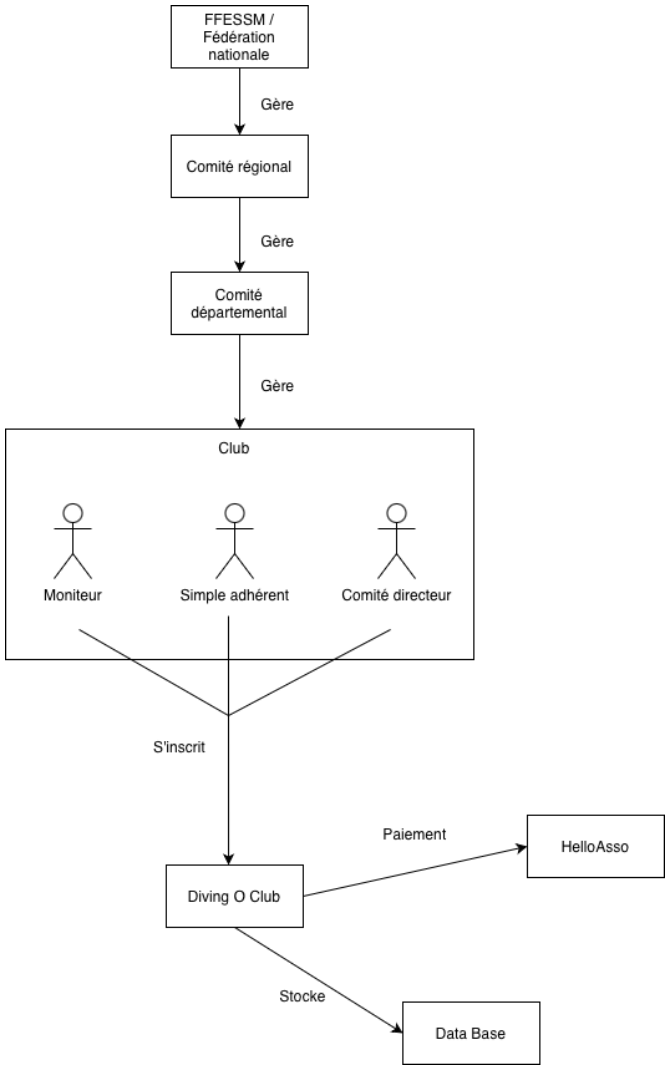


Figure 1.1 – Diagramme de contexte du projet

1.3 Liens utiles

— GitHub About : https://docs.github.com/
— SMART Goals : https://bit.ly/smart-goals-atlassian
— Project Management Institute : https://www.pmi.org/
— Agile Manifesto : https://agilemanifesto.org/
— Business Model Canvas : https://bit.ly/business-model-canvas
— FFESSM — Bilan à 700 jours (paragraphe "Amélioration de notre système informatique") https://ffessm.fr/actualites/bilan-de-la-ffessm-a-700-jours
— Nombre de clubs de plongée en France : https://ffessm.fr/actualites/un-ete-immersif-pour-les-f

— Données sur l'âge des plongeurs en 2002 page 18 : https://ffessm.fr/uploads/media/docs/0001/02/9b74472404c10cd0bd3c09bf15f1a9b217159689.pdf
— Données sur l'âge des plongeurs en 2020 page 11 : https://www.dirm.mediterranee.developpement-durable.gouv.fr/IMG/pdf/mnca_2019_etudeplongee_version_finale.pdf

Chapitre 2
Cadrage et cahier des charges

2.1 Objectifs métier, techniques et pédagogiques

Cette section présente les objectifs principaux du projet *Diving O Club* selon trois axes complémentaires : métier, technique et pédagogique. Ces objectifs guident l'ensemble du cadrage du projet et structurent les livrables jusqu'à la version finale.

Objectifs métier

Les objectifs métier répondent aux besoins concrets d'un club associatif de plongée fonctionnant avec des ressources bénévoles :

— **Centraliser l'ensemble de la gestion du club** : adhérents, événements, certificats, inscriptions et paiements, aujourd'hui dispersés dans plusieurs outils (Excel, documents papier, e-mails).
— **Réduire d'au moins 50% la charge administrative** liée aux inscriptions, relances, vérifications de documents et suivi des obligations fédérales.
— **Garantir une expérience utilisateur simple et accessible**, adaptée y compris aux adhérents peu familiers du numérique.

Ces objectifs sont mesurables à travers :

— un **taux d'adoption cible de 20 utilisateurs actifs** durant la phase bêta ;
— une **diminution de 50% des relances manuelles** dans les deux premiers mois ;
— un **niveau de satisfaction de 90%** sur le questionnaire de fin de bêta.

Objectifs techniques

Les objectifs techniques définissent le cadre technologique et les exigences de qualité logicielle :

— **Concevoir une architecture trois-tiers scalable** : Frontend (Next.js / React), API backend (Node.js / Nest.js) et base de données (PostgreSQL complété par MongoDB pour certains usages analytiques).
— **Mettre en place une authentification sécurisée** reposant sur JWT, cookies HttpOnly et un contrôle des accès par rôle (RBAC), en conformité avec les bonnes pratiques et le RGPD.
— **Assurer des performances maîtrisées** : disponibilité cible de 99 % en bêta et temps de réponse moyen inférieur à 300 ms pour 95 % des requêtes.
— **Intégrer des mécanismes de robustesse** : sauvegardes régulières, monitoring (logs MongoDB), journalisation des actions sensibles et préparation à l'isolation multi-tenant pour une future évolution multi-clubs.

Objectifs pédagogiques (CDA)

Le projet s'inscrit dans le cadre du titre *Concepteur Développeur d'Applications*. Il constitue une mise en situation professionnelle complète permettant de mobiliser l'ensemble des compétences visées :

— **Mettre en œuvre l'ensemble des compétences du titre CDA** dans un contexte réel : analyse, conception, développement, sécurisation, documentation.
— **Concevoir et documenter une architecture complète** : modèles de données (MCD/MLD), user stories, diagrammes, backlog, documentation API, CI/CD.

— **Développer et déployer une application sécurisée** incluant une gestion avancée des rôles et du traitement des données personnelles.
— **Interagir avec un client associatif réel** (club pilote), recueillir ses besoins, intégrer ses retours et adapter le développement en continu.

Ces objectifs forment un socle cohérent aligné avec les contraintes du projet (ressource unique, disponibilité limitée du comité, environnement associatif bénévole) et garantissent une base solide pour l'évolution future du produit.

2.2 Cibles et parties prenantes

Cette section identifie les différents types d'utilisateurs finaux de l'application ainsi que les parties prenantes impliquées dans le projet. Elle constitue une base essentielle pour comprendre les besoins, les contraintes et l'influence de chaque acteur sur l'évolution de la solution.

Types d'utilisateurs

Les utilisateurs du système se répartissent en quatre grandes catégories. Cette classification présente les rôles et usages principaux sans entrer dans le détail, lequel est développé dans les personae qui suivent.

1. **Nouveau utilisateur / sans club**
   Personne souhaitant découvrir la plongée ou rejoindre un club FFESSM. Premiers usages : création de compte, recherche de club, demande d'inscription.

2. **Adhérent (pratiquant plongée / apnée)**
   Licencié FFESSM, utilise l'application pour s'inscrire aux activités et les payer, gérer son certificat médical.

3. **Moniteur / encadrant**
   Encadrant FFESSM, responsable de l'organisation, du contrôle des certificats et du déroulement des activités.

4. **Membre du comité directeur**
   Président, trésorier, secrétaire ou responsable administratif. Supervise les adhésions, paiements, validations et données du club.

Personae détaillés

Les personae permettent de représenter plus finement des profils représentatifs des utilisateurs réels, en intégrant leurs motivations, irritants, comportements et besoins spécifiques. Deux personae ont été définis pour guider les choix de conception.

Persona 1 — Marc, 52 ans, Adhérent régulier (plongée bouteille)

Profil : Marc est adhérent depuis 7 ans dans un club FFESSM. Il travaille dans le bâtiment et utilise peu les outils numériques. Il consulte son téléphone surtout pour ses mails et WhatsApp. Il participe régulièrement aux sorties techniques et aux entraînements piscine.

Motivations :
— S'inscrire facilement aux entraînements et sorties.
— Avoir un suivi clair de son certificat médical.
— Payer facilement depuis son téléphone les sorties.

Frustrations / irritants :
— Difficulté à retrouver les informations éparpillées (Drive, mails, PDF).

— Manque d'indications sur les étapes à suivre pour certaines démarches.
— Perte de temps lorsqu'il n'arrive pas à s'inscrire ou que son certificat n'est pas à jour.

Besoins numériques :

— Interface simple, lisible, mobile-first.
— Parcours guidé pour l'inscription aux événements.
— Notifications claires en cas de certificat expiré ou dossier incomplet.

Objectifs personnels :

— Pratiquer la plongée sans stress administratif.
— Pouvoir s'inscrire rapidement et ne rien oublier.
— Être autonome, même sans grande maîtrise informatique.

Persona 2 — Sophie, 38 ans, Encadrante (E2) et membre du comité

Profil : Sophie est monitrice fédérale (E2) et membre du comité depuis 3 ans. Elle organise chaque mois des sorties techniques, gère les validations de certificats et les listes d'inscrits. Elle utilise régulièrement son ordinateur et son smartphone, et recherche de l'efficacité.

Motivations :

— Gagner du temps dans la gestion administrative du club.
— Suivre en un coup d'œil les certificats, paiements et inscriptions.
— Avoir un outil fiable pour éviter les erreurs critiques lors des sorties.

Frustrations / irritants :

— Qualité et fiabilité inégales des documents envoyés (PDF, photos).
— Dossiers incomplets, certificats expirés, relances manuelles répétitives.
— Outils dispersés rendant difficile la coordination entre encadrants.

Besoins numériques :

— Tableau de bord clair pour suivre les adhérents et leurs statuts.
— Centralisation des certificats, inscriptions et paiements.
— Filtrage, notifications automatiques et système anti-erreur.

Objectifs professionnels :

— Assurer la sécurité des pratiquants.
— Réduire le temps administratif pour se concentrer sur l'encadrement.
— Avoir un outil fiable pour la coordination avec le comité.

Parties prenantes : analyse influence / intérêt

La réussite du projet Diving O Club dépend de différentes parties prenantes ayant des niveaux d'influence et d'intérêt variés. Leur identification permet d'adapter la stratégie d'implication et de communication tout au long du projet.

Liste des parties prenantes

— **Adhérents** : utilisateurs finaux, concernés par la simplicité d'usage et la clarté des parcours.
— **Moniteurs / encadrants** : acteurs opérationnels dont l'usage quotidien conditionne l'adoption interne.
— **Comité directeur** : décideurs métier, garants de la conformité et des priorités internes.
— **Développeur** : responsable technique du projet, conception, qualité, sécurité et maintenance.
— **HelloAsso** : service tiers pour les paiements en ligne, intégré au fonctionnement administratif.

Analyse influence / intérêt

Cette analyse permet d'identifier les acteurs stratégiques et de hiérarchiser les besoins :

— **Forte influence, fort intérêt** : comité directeur, développeur. Ils orientent les priorités, arbitrent le MVP et conditionnent la qualité finale.
— **Influence moyenne, intérêt élevé** : moniteurs / encadrants. Ils influencent directement l'usage sur le terrain et l'adéquation aux pratiques du club.
— **Faible influence, intérêt moyen / élevé** : adhérents. Leur satisfaction détermine l'adoption à long terme, en particulier sur le mobile-first.
— **Influence faible, intérêt moyen** : HelloAsso. Fournit un service critique (paiements), mais sans impact direct sur la stratégie.

Impact sur le périmètre du MVP

L'analyse des parties prenantes permet également de définir avec précision les fonctionnalités qui doivent impérativement figurer dans le MVP. Les besoins critiques du comité directeur (certificats, paiements, inscriptions) et les irritants majeurs identifiés chez les adhérents (parcours complexe, manque de centralisation) ont orienté les priorités suivantes :

— parcours d'inscription simplifié pour les adhérents ;
— centralisation certificats–paiements–inscriptions pour le comité ;
— visibilité immédiate des inscrits et des documents pour les encadrants.

Ainsi, le MVP se concentre uniquement sur les modules apportant une valeur directe aux parties prenantes clés : authentification, gestion des activités, inscriptions, certificats médicaux et synchronisation des paiements.

Tableau d'analyse des parties prenantes

Partie prenante	Intérêt pour le projet	Influence	Commentaire
Comité directeur	Très élevé	Très élevé	Définit les priorités métier, valide les choix stratégiques.
Moniteurs / encadrants	Élevé	Moyen	Leur usage quotidien conditionne l'adoption opérationnelle.
Adhérents	Moyen	Faible	Leur expérience utilisateur détermine la satisfaction à long terme.
Développeur (Kevin)	Très élevé	Élevé	Garantit qualité technique, sécurité et livraison dans les délais.
HelloAsso	Moyen	Faible	Service indispensable au paiement, doit être surveillé.

Matrice pouvoir / intérêt et stratégies associées

La matrice pouvoir / intérêt permet de définir une stratégie d'implication adaptée à chacune des parties prenantes.

Partie prenante	Pouvoir	Intérêt	Stratégie de gestion
Comité directeur	Élevé	Élevé	Impliquer de près : revues bimensuelles, arbitrage des priorités, validation des étapes.
Moniteurs / encadrants	Moyen	Élevé	Consulter régulièrement : tests mensuels, retours sur l'ergonomie et les parcours.
Adhérents	Faible	Moyen	Informer : parcours guidé, interface simple, tutoriels, feedback utilisateur en continu.
HelloAsso	Faible	Moyen	Surveiller : tests sandbox, plans de reprise en cas d'incident, monitoring des paiements.

Risques liés aux parties prenantes

L'analyse met également en lumière plusieurs risques qui peuvent impacter l'adoption du produit :

— **Adhérents peu technophiles** : risque de non-adoption si l'interface n'est pas suffisamment simple et mobile-first.

— **Encadrants surchargés** : faible disponibilité pour tester ou remonter des retours, ce qui peut ralentir l'itération.

— **Comité directeur** : risque de surcharge administrative si la synchronisation paiements–inscriptions n'est pas fiable.

— **HelloAsso** : dépendance externe ; un changement d'API ou une indisponibilité peut bloquer certaines opérations.

La roadmap et le périmètre du MVP intègrent ces risques afin de garantir une adoption progressive, sécurisée et réaliste au sein du club pilote.

Conclusion

Cette analyse garantit une communication adaptée et une prise en compte fine des besoins des différents acteurs. Elle permet d'ajuster le périmètre du MVP, de construire une roadmap cohérente et d'assurer une adoption optimale par les utilisateurs stratégiques du club.

2.3 Exigences fonctionnelles

2.3.1 Fonctionnalités « Front Office »

Front Office :

— Création de compte / authentification
— Création d'un club : un utilisateur peut proposer un nouveau club, qui sera placé en « attente de validation ».
— Suivi du statut de validation du club (en attente / validé / refusé).
— Gestion du profil utilisateur (coordonnées, licence, certificat médical)
— Calendrier des événements (consultation et inscription / modification / annulation)
— Paiement en ligne via HelloAsso pour les événements payants
— Campagne d'adhésion (cotisation) : formulaire HelloAsso, ajout au **panier**, paiement et attestation
— Panier (événements *et* items boutique/adhésion) : visualisation, suppression, paiement
— Panier pour consulter la liste des activités qu'il reste à payer
— Consultation de l'historique personnel (inscriptions, paiements, historique, certificats)
— Accès à la page du club

2.3.2 Fonctionnalités « Back Office »

Back Office :

— Gestion des utilisateurs (invitation, modification de profil, désactivation)
— Gestion des rôles (adhérent / moniteur / comité directeur / administrateur du club)
— Gestion des clubs :
  — validation des clubs nouvellement créés (admin technique)
  — attribution automatique du rôle « administrateur du club » au créateur
  — modification des informations du club
— Gestion des événements (création, modification, suppression, capacité, tarifs, règles)
— Validation des certificats médicaux
— Suivi des inscriptions (confirmées / annulées)
— Suivi des paiements (via HelloAsso + marquage manuel si besoin pour règlement par espèces ou chèque)
— Configuration du club (informations, page publique, visuels)

Spécification des fonctionnalités par couche

Cette section détaille les fonctionnalités du MVP en les répartissant entre les trois couches de l'architecture : **front-end**, **back-end** et **API REST**. Cette structuration permet de clarifier les responsabilités de chaque couche et de garantir une cohérence entre interface, logique métier et points d'accès.

Couche	Fonctionnalité	Détails / critères	API
Front	Authentification	Formulaire email + mot de passe, messages d'erreur, redirection tableau de bord	`POST /auth/login`
Front	Inscription	Formulaire nom, email, mot de passe (avec confirmation) ; validations côté client	`POST /auth/register`
Front	Création de club	Formulaire nom + email ; configuration initiale ; validation client	`POST /clubs`
Front	Recherche de club	Champ de recherche, liste filtrée, suggestions, affichage rapide	`GET /clubs?search=`
Front	Page d'un club	Affichage nom, description, encadrants, activités ; bouton "Rejoindre le club"	`GET /clubs/:id`
Back	Gestion clubs	Création tenant, assignation rôle owner, validations métier, récupération des infos club	`GET /clubs/:id`
API	Liste des clubs	Renvoie les clubs, avec filtre, tri et pagination	`GET /clubs`
Back	Gestion des rôles	Attribution des rôles, contrôle d'accès aux routes, journaux d'accès	(interne)
Back	Création d'événement	CRUD événement ; affichage dans le calendrier ; contrôles métier	`GET /events`, `POST /events`, `PATCH /events/:id`
Front	Inscription à un événement	Bouton actif si certificat valide et places disponibles ; messages d'état	`POST /registrations`
API	Paiements HelloAsso	Réception webhook ; état payé/non payé ; id transaction stocké	`POST /webhooks/helloasso`
Back	Certificat médical	Upload ; statuts en attente / validé / refusé ; historique des vérifications	`POST /certificates`, `PATCH /certificates/:id/status`
Front	Notifications	Bannières d'alerte et emails transactionnels ; préférences utilisateur	`POST /notifications/test`
Back	Administration minimale	Liste inscrits, validations, filtres, export simplifié	`GET /admin/overview`

User stories (format normé)

Les besoins fonctionnels du projet sont exprimés sous forme de **User Stories**, selon la structure standardisée issue des pratiques Agile :

En tant que [rôle], je veux [objectif] afin de [bénéfice].

Ce format unifié garantit une compréhension partagée entre les profils métier et techniques. Toutes les User Stories du projet sont rédigées selon cette norme et organisées dans GitHub Project (backlog complet : EPIC → US → tâches).

Exemple de User Story détaillée (US-00)

US-00 — Création de compte utilisateur

User Story

En tant qu'**utilisateur externe non inscrit**, je veux créer un compte via un formulaire simple (nom, email, mot de passe) afin d'accéder à mon espace personnel et rejoindre ou créer un club.

Critères d'acceptation (format Gherkin)

```
Scenario: Inscription réussie
  Given un utilisateur non inscrit
  When il saisit un nom, un email valide et un mot de passe sécurisé
  Then son compte est créé
  And il est automatiquement authentifié
  And il est redirigé vers son tableau de bord

Scenario: Email déjà utilisé
  Given un utilisateur non inscrit
  When il saisit un email déjà existant
  Then un message "Email déjà utilisé" apparaît
  And la création est refusée

Scenario: Mot de passe invalide
  Given un utilisateur non inscrit
  When il saisit un mot de passe ne respectant pas la politique
  Then un message d'erreur apparaît
```

Backend :

— Endpoint `POST /auth/register`
— Validation DTO (email unique, règles mot de passe)
— Hash bcrypt
— Création utilisateur en base
— Génération des tokens (access + refresh)
— Réponse JSON + refresh token en cookie `httpOnly`

Frontend :

— Formulaire (nom, email, mot de passe, confirmation)
— Validation en direct
— Affichage des erreurs
— Redirection tableau de bord

User Stories du MVP

L'ensemble des User Stories (US-01 à US-08) suivantes sont rédigées sur le même format normé et détaillées dans le **GitHub Project du MVP**. Elles couvrent :

— **US-01** : Connexion
— **US-02** : Création d'événement
— **US-03** : Inscription à une activité
— **US-04** : Gestion du certificat médical
— **US-05** : Paiement via HelloAsso
— **US-06** : Suivi administratif minimal
— **US-07** : Création de club
— **US-08** : Validation d'un club par l'admin technique

Chacune comporte :

— une formulation standardisée “En tant que… je veux… afin de…”,
— des critères d'acceptation Gherkin,
— et un découpage technique associé (tâches Backend/Frontend).

Mesure d'impact par fonctionnalité

Chaque fonctionnalité du MVP est associée à un indicateur mesurable (KPI) permettant d'évaluer son impact réel sur l'organisation du club pilote. Ces indicateurs complètent les objectifs SMART et permettent un suivi opérationnel concret.

Fonctionnalité	KPI associé	Méthode de mesure
Authentification / création de compte	Taux de complétion du parcours	Logs de création de compte ; % d'utilisateurs allant jusqu'à la fin du formulaire.
Création / gestion des clubs	Nombre de clubs créés dans le mois suivant la Beta	Analyse des enregistrements `Club` et des rôles associés.
Inscriptions aux événements	Taux d'inscriptions enregistrées via la plateforme (>80%)	Comparaison inscriptions plateforme vs listes WhatsApp/Excel.
Certificats médicaux	Réduction des certificats expirés non détectés (objectif : -70%)	Logs de validation, indicateurs avant/après déploiement.
Synchronisation HelloAsso	Taux d'erreurs dans la correspondance paiement/inscription (objectif : <5%)	Analyse journalière des statuts paiement vs inscription.
Calendrier / événements	Nombre de connexions au calendrier et inscriptions issues du planning	Statistiques d'affichage `/events` et des clics sur "S'inscrire".
Administration comité	Réduction du temps administratif hebdomadaire (objectif : -50%)	Auto-déclaration + mesure du temps passé avant/après via journal hebdo.

Ces indicateurs permettent d'évaluer objectivement l'impact du MVP sur le fonctionnement du club pilote et d'orienter les priorités pour la version Beta et la V1.

2.3.3 Découpage Front / Back (fonctionnalités clés)

Fonction	Front (Next.js/React)	Back (Nest.js)
Auth / Sessions	Formulaires, états UI, garde routes	Endpoints /login, /refresh, /logout ; JWT, cookies httpOnly
Événements	Liste, détail, formulaires CRUD	Endpoints /events (CRUD), règles capacité
Inscriptions	Bouton s'inscrire/annuler, vues perso	/registrations, contrôle certificat/-capacité
Certificats	Upload, aperçu statut	/certificates (upload, statut), validations comité
Paiements	Redirection, retour statut	Webhooks/retour HelloAsso, persistance statut
Admin minimal	UI listes (users, events, validations)	Endpoints admin sécurisés + logs

2.3.4 L'utilisateur (public)

Les types d'utilisateurs : Rôles et permissions utilisateurs :

- **Utilisateurs sans club (externes)** : peuvent compléter leur profil, créer un club, rechercher et voir les clubs, faire une demande pour rejoindre un club, ajouter leur certificat médical, consulter leur panier et leur historique de paiement.
- **Adhérents (internes)** : participent aux activités du club, accèdent au calendrier, s'inscrivent et paient en ligne.

— **Moniteurs (internes opérationnels)** : consultent les inscrits, contrôlent les certificats et gèrent les événements.
— **Comité directeur (administrateurs métier)** : gèrent les utilisateurs, les paiements, les documents administratifs et valident les certificats médicaux.
— **Développeur / administrateur technique** : gère la configuration technique, la sécurité, les mises à jour et la maintenance de l'application.

2.3.5 Confidentialité

L'application Diving O Club traite des **données à caractère personnel** dans le cadre du fonctionnement des clubs FFESSM. Elle applique strictement les principes du **RGPD** : minimisation, finalité déterminée et explicite, transparence, limitation de la durée de conservation, sécurité renforcée et respect des droits utilisateurs (*accès, rectification, suppression*).

Exigences de confidentialité et conformité RGPD

Le projet respecte les obligations de protection des données : **chiffrement**, **journalisation**, contrôle d'accès (**RBAC**), **chiffrement en transit via HTTPS** et conservation limitée. Le tableau ci-dessous récapitule les données traitées dans le MVP, leur finalité, leur base légale et les mesures de sécurité associées.

Données collectées	Finalité	Base légale	Durée de conservation	Mesures de sécurité
Nom, prénom, email	Création du compte adhérent ; communication interne du club	Exécution du contrat / intérêt légitime	Suppression 12 mois après inactivité ou demande de l'utilisateur	Stockage chiffré, accès restreint, journalisation des accès
Mot de passe (hashé)	Authentification et contrôle des accès	Exécution du contrat	Suppression immédiate à la suppression du compte	Hash bcrypt, aucune donnée en clair, politiques de mot de passe fortes
Certificat médical (PDF/JPG)	Vérification obligatoire de l'aptitude à la pratique	Obligation réglementaire / sécurité des personnes	1 an après expiration du certificat ou suppression du compte	Stockage restreint, accès comité/encadrants uniquement, logs de consultation
Historique d'inscriptions aux événements	Gestion des activités et traçabilité interne	Intérêt légitime du club	3 ans (obligations associatives standard)	Sécurisation DB, contrôle RBAC, journalisation
Paiements HelloAsso (références, statuts)	Synchronisation paiement / inscription ; justificatifs comptables	Obligation comptable / exécution du contrat	10 ans (obligation comptable)	Webhook sécurisé, vérification de signature, logs anti-fraude
Logs d'accès (dates, IP approximative, actions)	Sécurité, détection d'anomalies, audit	Intérêt légitime (sécurité)	6 mois (conformément recommandations CNIL)	Stockage séparé, rotation, analyses d'accès suspects

Ces exigences garantissent la conformité du MVP et constituent une base pour le **registre des traitements** et la future documentation DPO.

2.3.6 Droits d'accès

Droits d'accès :

2.3.7 Authentification

Système d'authentification :

Table 2.1 – Matrice de permissions par rôle — Diving O Club

Fonctionnalité / Action	Adhérent	Moniteur	Comité directeur	Admin technique
Créer un compte / se connecter	V	V	V	—
Modifier son profil	V	V	V	—
Voir le calendrier des événements	V	V	V	—
S'inscrire / annuler une inscription	V	V	V	—
Payer via HelloAsso	V	V	V	—
Uploader un certificat médical	V	V	V	—
Valider / refuser certificats	Lecture	Lecture	V (validation)	—
Créer / modifier / supprimer un événement	X	V	V	—
Consulter les participants	V	V	V	—
Gérer les utilisateurs (ajout / rôle / désactivation)	X	X	V	—
Suivi des paiements et statut	X	X	V	—
Configuration du club / documents internes	X	X	V	—
Paramètres techniques / intégrations / logs	X	X	X	V

Authentification et sécurité :

- **Connexion** : par email et mot de passe, avec validation des identifiants et contrôle de format (mot de passe conforme à une politique de sécurité via REGEX).
- **Sessions** : authentification stateless via JWT d'accès à durée courte, renouvelé par un refresh token stocké en cookie sécurisé (`httpOnly`). Expiration automatique et révocation en cas de déconnexion ou de réinitialisation.
- **Sécurité des comptes** :
  - mots de passe hachés (bcrypt)
  - protection anti-bruteforce
  - possibilité future d'ajouter une authentification à deux facteurs (2FA)
- **Récupération de compte** : procédure par email sécurisé (lien signé à expiration limitée), invalidation des anciennes sessions et limitation de l'utilisation du lien.
- **Autorisation par rôle** : accès conditionné par le rôle (adhérent / moniteur / comité directeur), contrôle des droits effectué côté back-end sur chaque route sensible.

2.4 Exigences et choix techniques

2.4.1 Exigences

- **Performance :** Temps de réponse moyen < 300 ms sur les endpoints critiques (authentification, événements, inscriptions, paiements) ; support d'au moins 100 utilisateurs simultanés par club.
- **Sécurité :** HTTPS obligatoire, mots de passe hachés (bcrypt), authentification JWT (accès + refresh httpOnly), contrôle d'accès basé sur les rôles (RBAC), journalisation des actions sensibles (certificats, paiements, création/gestion de club).
- **Disponibilité & résilience :** Objectif de 99% d'uptime en phase bêta, sauvegardes régulières de la base PostgreSQL, reprise possible en cas d'échec de paiement HelloAsso.

— **Scalabilité :** Architecture trois-tiers (Front Next.js / Back Nest.js / Base de données), découplée et multi-tenant (`club_id`) ; possibilité d'ajouter de nouveaux clubs sans refonte structurelle.
— **Maintenabilité :** Code organisé par modules (Nest.js), services, contrôleurs, DTOs, validation côté serveur (Pipes), documentation OpenAPI et pipeline CI/CD avec tests automatisés.
— **Confidentialité & conformité :** Données sensibles chiffrées, contrôle d'accès strict, conformité RGPD (consentement, droit d'accès, droit à l'effacement, limitation des données collectées).

2.4.2 Choix

Les choix techniques et leurs justifications objectives sont consignés dans le fichier DECISIONS.md du dépôt. Chaque entrée précise contexte, alternatives et critères de mesure.

Analyse des alternatives techniques

Au-delà du tableau comparatif ci-dessus, plusieurs solutions alternatives ont été étudiées afin d'évaluer leur pertinence pour un projet associatif multi-tenant et conforme aux exigences définies (performance, RGPD, maintenabilité, coûts).

Backend / BaaS. Des services managés comme **Firebase** ou **Supabase** ont été envisagés pour accélérer le développement. Ils ont été écartés pour éviter le *lock-in* fournisseur, les coûts variables d'usage, et les limites de requêtage (Firestore peu adapté aux relations complexes : inscriptions, certificats, paiements). Nest.js permet au contraire une maîtrise totale de la structure, des règles de sécurité (guards RBAC) et du schéma multi-tenant.

Base de données. MySQL et **Firestore** ont été analysés comme alternatives à PostgreSQL. MySQL a été écarté en raison d'un support moins riche pour les contraintes d'intégrité avancées (nécessaires pour la cohérence événements/inscriptions/paiements). Firestore a été écarté pour ses limitations en transactions complexes et pour la difficulté à garantir la cohérence métier d'un club sportif. PostgreSQL s'est imposé pour sa robustesse OLTP et son outillage mature.

Journalisation. La journalisation dans **PostgreSQL JSONB** a été envisagée mais non retenue, afin de ne pas mélanger charge métier et volumétrie des logs. Des solutions lourdes comme ELK ont été écartées par souci de simplicité opérationnelle. **MongoDB** offre un bon compromis léger, souple et indépendant.

Paiement. Des solutions plus génériques comme **Stripe** ont été étudiées, mais **HelloAsso** s'est révélé plus adapté au contexte associatif français : gestion des campagnes d'adhésion, reçus automatiques, conformité aux pratiques des clubs FFESSM.

Authentification. Des solutions tierces (Auth0, Firebase Auth, Clerk) n'ont pas été retenues pour des raisons de coût, de dépendance et de contrôle limité sur les jetons. Les **sessions serveur** ont également été envisagées mais rejetées au profit d'un modèle **JWT + refresh httpOnly**, plus simple à scaler et mieux adapté à un front Next.js.

Cette analyse garantit que les choix retenus ne sont pas arbitraires mais fondés sur une évaluation comparative cohérente avec les contraintes du projet et du contexte associatif.

Frontend — Next.js / React (mobile-first)

— **Pourquoi Next.js ?** Framework complet, support SSR/SSG si nécessaire, très bon support PWA, performances optimisées, composants réutilisables, accessibilité accrue.
— **Alternatives envisagées :** React seul (moins structuré, moins scalable).
— **Impact exigences :** Parcours mobile-first fluide pour publics non technophiles, interface optimisée, itérations rapides.

Backend — Node.js / Nest.js

— **Pourquoi Nest.js ?** Architecture modulaire (modules, services, contrôleurs), fortement typée, maintenable, testable ; logique métier centralisée dans les services ; DTOs et pipes de validation ; guards intégrés pour le RBAC.
— **Alternatives :** Express (plus léger mais moins structuré pour un projet de cette taille).
— **Impact exigences :** Temps de réponse < 300ms, validation stricte côté serveur, sécurité renforcée, maintenance facilitée.

Données — PostgreSQL + MongoDB (hybride)

— **PostgreSQL (données métiers) :** Cohérence transactionnelle (inscriptions, paiements, certificats, clubs, rôles), intégrité référentielle, relations claires.
— **MongoDB (logs / traçabilité) :** Stockage flexible pour journaux, audit trail, rapports et snapshots.
— **Pourquoi deux bases ?** Optimisation selon le type de données : OLTP relationnel (PostgreSQL) + logs/documentation (MongoDB).
— **Impact exigences :** Séparation des charges, meilleure scalabilité multi-club, performance sur les écritures.

Comparatif synthétique (alternatives considérées)

Le tableau ci-dessous présente les choix techniques retenus, les principales alternatives envisagées, ainsi que la justification de la décision. Ce comparatif répond aux exigences du référentiel RNCP concernant l'analyse des alternatives.

Choix retenu	Alternative	Justification du choix	Pourquoi l'alternative n'a pas été retenue
PostgreSQL	MySQL	Meilleure gestion des types avancés (JSONB, géodonnées), contraintes fortes et transactions.	Support plus limité des relations complexes ; moins adapté au futur module « spots de plongée ».
Nest.js (Node)	Express.js	Architecture modulaire, sécurisée, testable, inspirée d'Ecto (expérience Phoenix).	Express requiert beaucoup de « plomberie », pas de structure native, dette technique plus forte.
Next.js (React)	React seul	Framework structuré, SSR/ISR, PWA-ready, meilleures performances et SEO.	Moins structuré, nécessiterait de réimplémenter du routing et de la sécurité.
MongoDB	Firestore	Flexible pour les logs, coûts maîtrisés, intégration simple Node.js.	Coûts variables, verrou fournisseur, limites transactionnelles sur Firestore.
HelloAsso	Stripe	Solution gratuite, optimisée pour les associations françaises FFESSM.	Frais Stripe, complexité comptable, moins aligné avec les pratiques des clubs.
JWT + refresh	Sessions serveur	Stateless, scalable, idéal pour Next.js + API Nest.	Sessions server-side compliquent la scalabilité (sticky sessions).

Analyse des alternatives techniques

Le choix des technologies retenues pour Diving O Club a été précédé d'une évaluation objective d'alternatives. Le tableau précédent synthétise les options étudiées, ainsi que les critères qui ont conduit à retenir les solutions finales. Les décisions ont été motivées par la maintenabilité, la sécurité, la conformité RGPD, la scalabilité et l'adéquation au contexte associatif.

Frontend : Next.js + React vs React seul React seul aurait simplifié l'architecture (SPA), mais l'absence de SSR/ISR, de routing structuré et d'optimisations intégrées aurait complexifié le développement long terme. Next.js apporte un cadre plus robuste, une meilleure performance perçue et une architecture mieux adaptée à un projet évolutif.

Backend : Nest.js vs Express.js Express est minimal et rapide à mettre en place, mais impose de structurer manuellement l'ensemble de l'application (middleware, validation, architecture). Pour un projet multi-tenant avec logique métier complexe, Nest.js offre un cadre beaucoup plus sûr et modulaire (DI, modules, pipes, guards), réduisant la dette technique et les risques de sécurité.

Base de données métier : PostgreSQL vs MySQL MySQL a été envisagé, mais PostgreSQL offre des garanties plus fortes en termes de contraintes relationnelles, de typage, de transactions et de gestion avancée du JSON (JSONB). Ces caractéristiques sont indispensables pour gérer les relations clubs $\rightarrow$ événements $\rightarrow$ inscriptions $\rightarrow$ certificats.

Logs et audit : MongoDB vs PostgreSQL JSONB Stocker les logs dans PostgreSQL était possible, mais aurait pénalisé les performances de la base métier. MongoDB permet une grande flexibilité dans la structure des journaux (schémas variables), un volume important d'écritures et une séparation claire entre données critiques et données techniques (audit RGPD).

Paiement : HelloAsso vs Stripe Stripe est plus complet mais non optimisé pour les associations françaises (part commissions, justificatifs fiscaux, modèle légal). HelloAsso fournit un flux parfaitement adapté au contexte associatif, sans frais et avec un écosystème pensé pour le bénévolat et les clubs.

Authentification : JWT + refresh vs sessions serveur Les sessions côté serveur auraient simplifié la logique, mais auraient posé des contraintes de scalabilité (stockage d'état, sticky sessions). Le couple JWT + refresh httpOnly permet une architecture stateless plus adaptée au front-end Next.js, tout en assurant une sécurité renforcée via rotation des tokens et séparation accès/refresh.

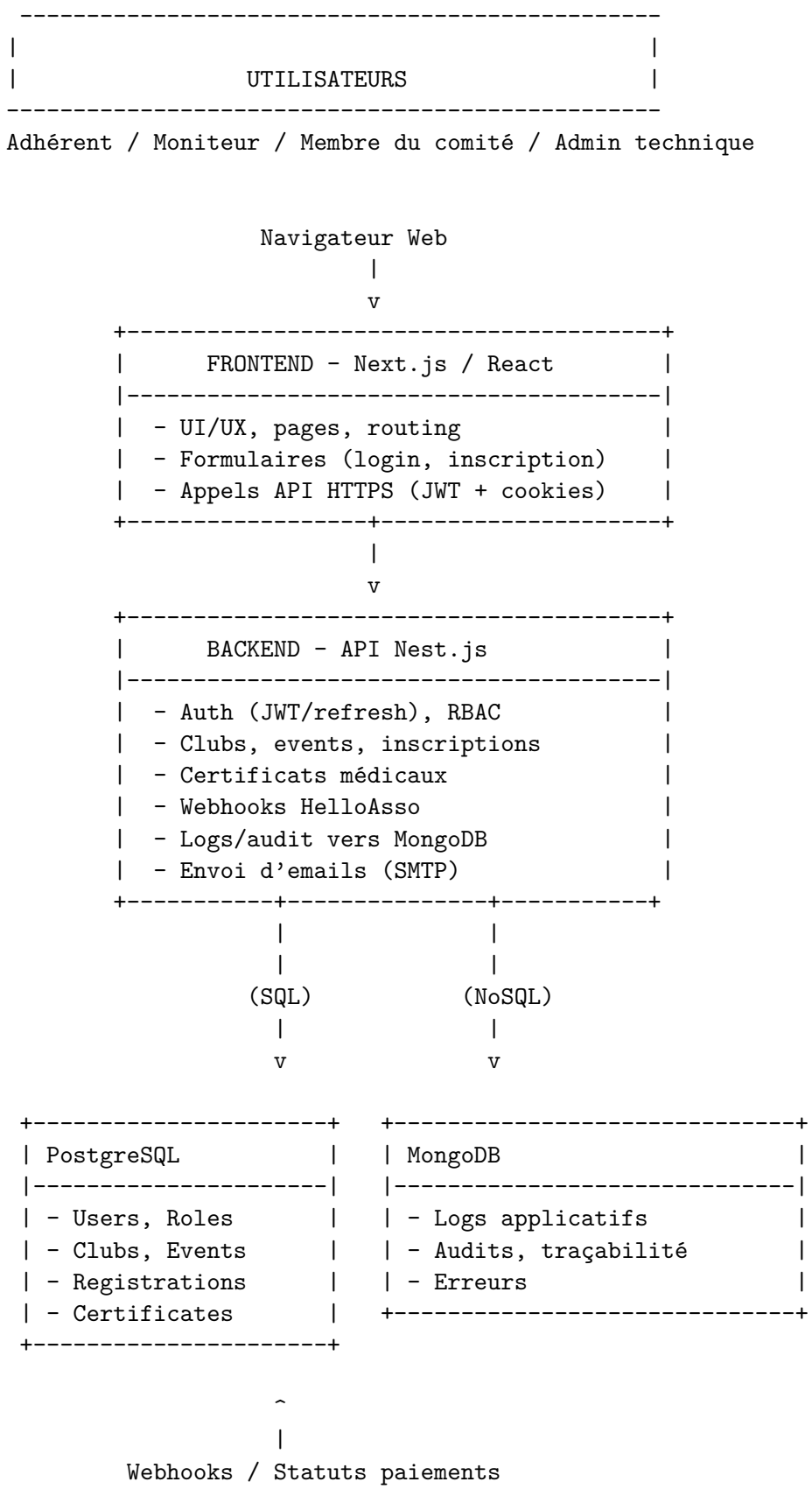
Authentification & sécurité

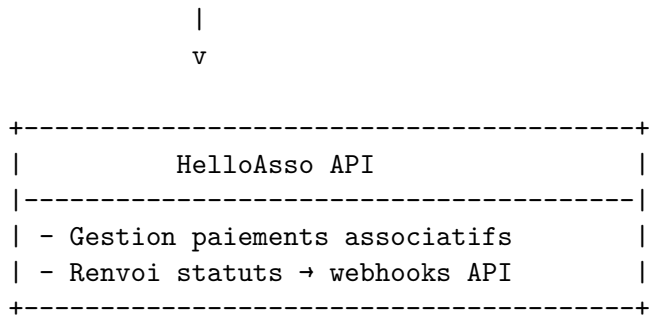
- JWT accès + refresh cookie `httpOnly`
- RBAC (adhérent / moniteur / comité / admin du club / admin technique)
- Hash bcrypt, rate limiting, CORS/TLS, guards Nest.js
- Contrôles de validation via DTO + Pipes
- **Pourquoi ?** Standards éprouvés, sécurisés, testables dans CI/CD.

Architecture 3-tiers — responsabilités & flux

```
[Client Web Next.js/React]
     |  HTTPS (JWT accès + cookie refresh httpOnly)
     v
[API Node/Nest.js]
     |-- RBAC, validation DTO/Pipes, logs Mongo
     |-- Intégrations: HelloAsso
     v
[PostgreSQL]  (users, roles, clubs, events, registrations, payments, certificates)
[MongoDB]     (logs, traçabilité, audits, snapshots)
```

Modèle C4 — Niveau Container (Vue des principaux blocs exécutables)





Paiements — HelloAsso

— **Pourquoi ?** Solution dédiée aux associations françaises, reçus automatiques, workflow paiements simple.
— **Alternatives :** Stripe (plus générique, moins adapté aux clubs FFESSM).

Scalabilité & évolutivité

— Multi-tenant via `club_id`
— Cache applicatif (post-MVP)
— Extensions prévues : 2FA, carnet de suivi, boutique plus complète, évolution Next.js SSR

Maintenabilité & qualité

— Modules Nest.js (users, auth, clubs, events, payments, certificates)
— Tests API des endpoints critiques
— Linting / formatage / documentation OpenAPI
— Pipeline CI/CD (tests → build → déploiement)

2.5 Définition du MVP

Méthode de priorisation MoSCoW

Afin de définir le périmètre fonctionnel prioritaire, la méthode MoSCoW a été utilisée. Cette approche, couramment utilisée en gestion de projet agile, permet de classer les fonctionnalités selon leur niveau de criticité pour la première version du produit.

Les priorités ont été établies en tenant compte :
— des besoins réels identifiés auprès du club de plongée (entretiens avec le président, les moniteurs et les adhérents)
— des contraintes associatives (bénévolat, temps limité, simplicité d'usage)
— des objectifs métier (réduction de la charge administrative, fiabilité du suivi)
— des capacités de développement dans le cadre du projet CDA (une seule ressource, planning défini)

Cette analyse garantit que la première version (MVP) se concentre uniquement sur les fonctionnalités indispensables à l'utilisation opérationnelle du système en conditions réelles, à savoir créer un compte, se connecter, rechercher un club et afficher la page du club.

Must Have

Fonctionnalité	Justification / Raison du classement
Authentification et rôles (adhérent / moniteur / comité)	Condition indispensable pour accéder aux données personnalisées et au multi-tenant. Bloque toutes les autres fonctionnalités.
Création de club par un utilisateur externe	Prérequis fondamental : sans club créé, aucune gestion d'adhérents, d'événements ou de certificats n'est possible.
Gestion des adhérents	Fonction coeur du projet. Sans elle, aucun suivi ni inscription possible. Valeur utilisateur immédiate.
Gestion des événements (création, édition, calendrier)	Fonction essentielle au fonctionnement du club : toute l'organisation repose sur ce module.
Inscriptions aux événements	Indispensable pour permettre aux adhérents de participer aux activités. Forte valeur utilisateur.
Paiements avec HelloAsso	Nécessaire pour automatiser les activités payantes et réduire les erreurs de gestion.
Gestion des certificats médicaux (upload + validation)	Exigence légale pour la pratique de la plongée. Priorité réglementaire incontournable.

Should Have

Fonctionnalité	Justification / Raison du classement
Emails / notifications automatiques	Réduction des relances manuelles. Améliore le confort du comité mais non bloquant pour la V1.
Tableau de bord simplifié (suivi inscriptions / paiements / validations)	Apporte de la visibilité et facilite la gestion interne, mais le fonctionnement de base reste possible sans.

Could Have

Fonctionnalité	Justification / Raison du classement
Gestion du matériel du club	Fonction utile mais non critique pour le fonctionnement administratifs des activités. Peut être reportée.
Carnet de suivi numérique	Valeur ajoutée pour les adhérents mais hors périmètre prioritaire du CDA et du MVP.

Won't Have

Fonctionnalité	Justification / Raison du classement
Statistiques avancées et exports PDF	Fonctionnalités lourdes et non indispensables pour la V1. Prévu en phase 2.
Intégration FFESSM (structures régionales / départementales)	Hypothèse de travail intéressante mais pas réalisable dans la première version.

Définition du MVP :

MVP — Avril 2026

Le MVP correspond à la première version pleinement fonctionnelle permettant à un club d'utiliser l'application en conditions réelles. Il inclut : l'authentification et les rôles, la création de compte, la **création de club sur demande** (avec **validation manuelle** et attribution initiale des

droits au créateur), la gestion des événements, les inscriptions (**avec ajout automatique au panier**), la synchronisation des paiements via HelloAsso, ainsi que la gestion de la conformité (*dépôt et validation du certificat médical*). Ce périmètre couvre le module cœur indispensable avant l'intégration du front, des paiements réels et des optimisations prévues entre mai et juin 2026.

Scénarios essentiels du MVP :

1. Connexion utilisateur et gestion de session (accès sécurisé par rôle : adhérent / moniteur / comité)
2. Consultation du calendrier des événements (liste, détails, capacité, conditions d'accès)
3. Inscription et annulation à un événement (blocage si certificat absent ou expiré)
4. Paiement via HelloAsso pour les événements payants (statut de paiement synchronisé côté back-end)
5. Gestion administrative de base côté comité :
   — création / modification d'un événement
   — validation ou refus d'un certificat médical
   — suivi des inscrits et statuts de paiement

Critères de succès du MVP :

— L'ensemble des scénarios listés est utilisable sans support par au moins 20 membres du club testeur
— Réduction mesurée d'au moins 50% des relances administratives manuelles habituelles

KPIs produit (mesure d'impact)

— **Réduction des relances admin** : -50% (moyenne hebdo, comité)
— **Adoption bêta** : ≥ 20 utilisateurs actifs/mois (club pilote)
— **Temps moyen d'inscription** : < 2 min (du clic à confirmation)
— **Taux d'abandon paiement** : $< 15\%$ sur événements payants
— **Taux d'erreur certificat** : $< 5\%$ (uploads invalides/refusés)
— **Perf endpoints critiques** : $p_{95} < 300$ ms (auth, events, register, payment)

Scénarios essentiels & critères d'acceptation (MVP)

Les critères d'acceptation du projet sont rédigés en **syntaxe Gherkin** (*Given / When / Then*), afin d'être **vérifiables, non ambigus et testables**.

— de décrire clairement les comportements attendus ;
— de garantir une compréhension partagée entre les parties prenantes ;
— d'établir des critères d'acceptation objectifs et mesurables ;
— de faciliter l'automatisation future des tests fonctionnels.

Méthode de rédaction des critères d'acceptation

Chaque User Story du projet possède des critères d'acceptation rédigés en **syntaxe Gherkin (Given / When / Then)** et documentés dans les **Issues du GitHub Project**. Pour ne pas alourdir ce document, quelques exemples représentatifs sont présentés ci-dessous.

US-01 — Inscription (register) — Critères d'acceptation (Gherkin)

```
Scenario: Inscription réussie
  Given un utilisateur non inscrit
  When il poste POST /auth/register avec name, un email valide
       et un mot de passe conforme à la politique
  Then l'API répond 201 Created avec un corps JSON contenant userId, email, name (sans passw
  And un enregistrement User existe en base avec email_verified = false
       et created_at non nul
  And un email de vérification est envoyé à l'adresse fournie
       (événement email.verification.requested loggé)
  And aucune session persistante n'est créée tant que l'email n'est pas vérifié
       (pas de cookie refresh)

Scenario: Email déjà utilisé
  Given un utilisateur non inscrit
  When il poste POST /auth/register avec un email déjà existant
  Then l'API répond 409 Conflict avec code = "EMAIL_TAKEN"
  And aucun nouvel utilisateur n'est créé en base
  And aucun email de vérification n'est envoyé

Scenario: Mot de passe invalide
  Given un utilisateur non inscrit
  When il poste POST /auth/register avec un mot de passe ne respectant pas la politique
  Then l'API répond 400 Bad Request avec code = "WEAK_PASSWORD"
       et la liste des règles violées
  And aucun utilisateur n'est créé
  And aucune session ni email n'est généré
```

Plan de validation utilisateur du MVP

Afin de garantir que le MVP répond aux besoins réels du club et des utilisateurs finaux, une session de validation sera organisée avec un panel représentatif (adhérents, moniteurs, membres du comité). Cette étape permet de tester l'ergonomie, la compréhension des parcours, la cohérence des informations et la fiabilité des fonctionnalités essentielles.

Objectifs de la session de test

- Valider que les parcours principaux (connexion, inscription, certificat, paiement) sont compréhensibles sans assistance.
- Identifier les frictions, incompréhensions ou lenteurs dans l'usage réel.
- Recueillir les retours des utilisateurs afin d'améliorer la version Beta.
- Confirmer que le MVP est utilisable en conditions de club.

Panel utilisateur

- **3 adhérents** dont 1 peu technophile.
- **2 moniteurs / encadrants**.
- **1 membre du comité** (trésorière ou secrétaire).

Fonctionnalités testées

- Connexion / création de compte.

— Consultation du calendrier des plongées.
— Inscription à un événement.
— Upload et validation du certificat médical.
— Paiement HelloAsso et synchronisation du statut.
— Accès au tableau de bord utilisateur.

Scénarios de validation

Les tests suivront un protocole basé sur des scénarios réels :
— **T1** : Un adhérent s'inscrit à une plongée, vérifie la place restante et valide son certificat.
— **T2** : Un moniteur consulte les inscrits, vérifie les certificats et contrôle les paiements.
— **T3** : Un adhérent peu technophile tente de naviguer seul sur le site (mobile-first).
— **T4** : Un membre du comité utilise l'interface d'administration minimale.

Méthode de recueil des feedbacks

— Observation directe (durée, erreurs, blocages).
— Questionnaire de satisfaction court (5 questions fermées + 1 ouverte).
— Notation de l'effort perçu (User Effort Score).
— Compte rendu synthétique intégré dans le GitHub Project (tickets d'amélioration).

Livrables attendus

— Fiche de test complétée pour chaque utilisateur.
— Synthèse des problèmes rencontrés.
— Liste des actions correctives prioritaires (MoSCoW).
— Ajout des améliorations dans le backlog Beta.

Calendrier prévisionnel

— Semaine 1 : Préparation des scénarios et du protocole.
— Semaine 2 : Session de test (2h, club pilote).
— Semaine 3 : Analyse des retours et intégration des correctifs dans la roadmap Beta.

Revue et validation des exigences avec les utilisateurs métier

Dans le cadre du projet, une première **validation réelle des exigences** a été réalisée avec les parties prenantes du club pilote (Aquaclub21). Lors de la réunion du comité directeur du **15 février 2025**, les besoins métier suivants ont été confirmés et priorisés :
— centralisation certificats–paiements–inscriptions ;
— simplification du parcours d'inscription adhérent ;
— visibilité immédiate des inscrits pour les encadrants ;
— importance du mobile-first pour les adhérents peu technophiles.

Un **compte rendu de réunion** (PDF) ainsi qu'une mise à jour du **GitHub Project** (section *Issues* → *Validation métier*) assurent la traçabilité de cette validation.

Afin de garantir que les fonctionnalités du MVP correspondent fidèlement aux besoins du club, une revue d'exigences sera organisée avec les parties prenantes métier (comité directeur, moniteurs, adhérents représentatifs). Cette revue permet de valider collectivement le périmètre, les priorités et les critères d'acceptation avant le développement.

Objectifs de la revue

— Vérifier que les exigences fonctionnelles couvrent bien les besoins du club.
— Identifier les oublis, ambiguïtés ou conflits entre exigences.
— Aligner les priorités (MoSCoW) avec les utilisateurs métier.
— Confirmer la cohérence entre le MVP, les user stories et les parcours utilisateurs.

Participants

— 1 à 2 membres du comité (président / trésorière).
— 1 moniteur encadrant (usage opérationnel terrain).
— 1 adhérent représentatif (peu technophile).
— Développeur / porteur du projet (facilitateur).

Déroulé de la session (1h30)

— Présentation synthétique du périmètre MVP.
— Lecture collaborative des user stories clés (US-01 à US-08).
— Vérification des critères d'acceptation (Given / When / Then).
— Revue de la matrice MoSCoW pour ajuster les priorités.
— Collecte des retours (questions, frustrations, risques perçus).
— Mise à jour du backlog sur GitHub Project.

Livrables produits

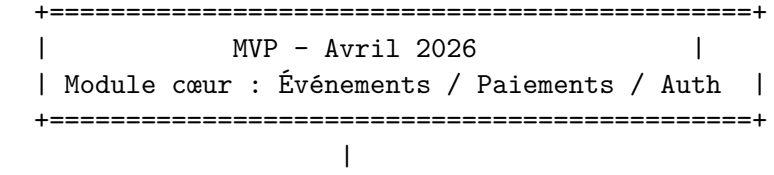
— Compte rendu de revue (PDF) : décisions, modifications, arbitrages.
— Mise à jour du backlog (priorités revues et exigences clarifiées).
— Liste des points ouverts / questions à trancher pour la Beta.
— Validation formelle des parties prenantes (signature numérique ou confirmation écrite).

Calendrier prévisionnel

— **J+0** : Envoi des exigences et des user stories aux participants.
— **J+7** : Session de revue (club pilote).
— **J+10** : Intégration des retours dans le backlog GitHub.

Bornes techniques minimales (V1)

— **Modèles V1 uniquement** : Users, Roles, Events, Registrations, Payments, MedicalCertificates.
— **Paiement** : parcours carte via HelloAsso (pas de remboursement ni d'export comptable).
— **Sécurité** : JWT accès court + refresh cookie httpOnly, RBAC simple (3 rôles).
— **Journalisation** : logs d'actions sensibles (validation certificat, création d'événement).
— **Admin** : back-office minimal (liste utilisateurs, événements, validations).
Utilisateur → **Connexion** → **Inscription** → **Certificat** → **Paiement** → **Confirmation**
Diagramme — MVP (Avril 2026) :



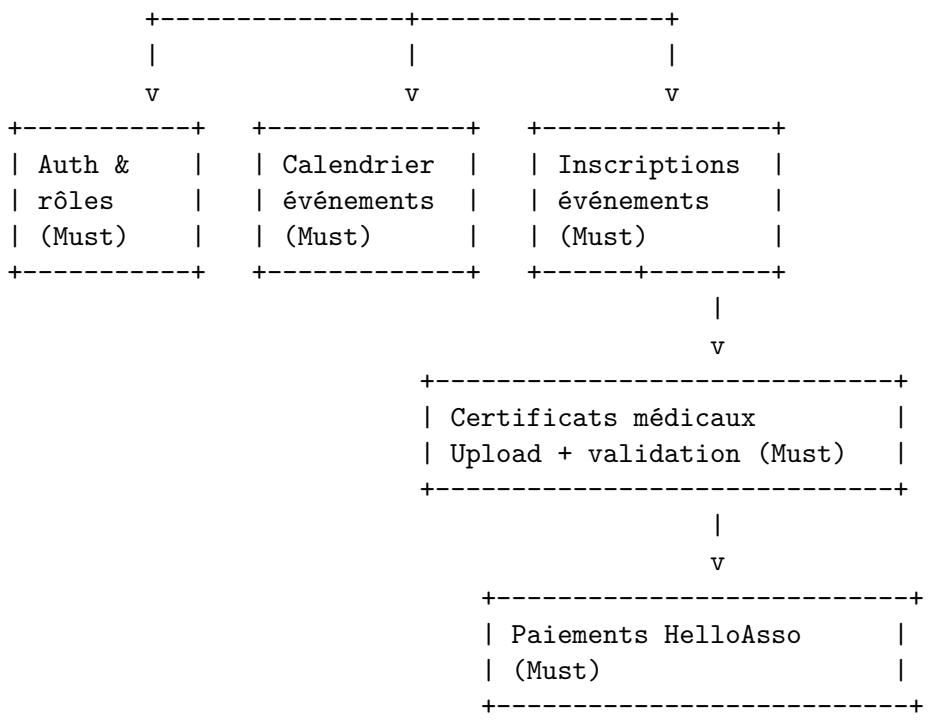
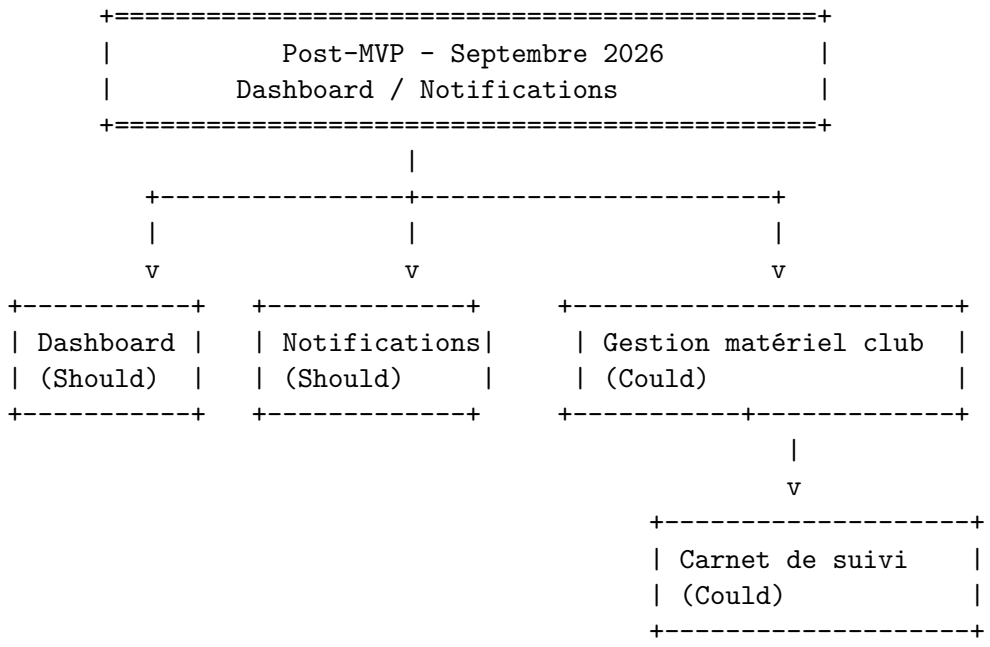


Diagramme MVP v1.1 (post-V1) :



Plan de validation avec utilisateurs (club pilote)

- **Ateliers cadrage (Semaine 1)** : 1h avec comité (priorités), 30min avec 2 moniteurs (contraintes terrain)
- **Tests exploratoires (mensuels)** : 5 adhérents + 1 moniteur ; scénarios MVP ; recueil friction
- **Canaux feedback** : formulaire court après inscription/paiement ; salon Discord dédié
- **Revue bimensuelle** : démonstration au comité, décisions de priorisation (MoSCoW)
- **Critères de sortie bêta** : KPIs atteints 2 semaines consécutives ; zéro bug bloquant

Analyse des risques (probabilité / impact / plan d'action)

Risque	Prob.	Impact	Mitigation
Charge projet sous-estimée	Moyenne	Élevé	Scope strict MVP, jalons bimensuels, coupe des « Should » si dérive
Adoption faible (adhérents)	Moyenne	Moyen	Mobile-first, tutoriels, tests mensuels, simplification parcours
HelloAsso indisponible	Faible	Élevé	Mode secours : inscription « non payée » + validation manuelle
Bug critique prod	Faible	Élevé	Tests API CI, rollback rapide, sauvegardes quotidiennes
Perte de données	Très faible	Très Élevé	Backups PostgreSQL, restauration testée mensuellement, logs Mongo
RGPD (accès non autorisé)	Faible	Élevé	RBAC strict, logs d'accès, revue des permissions

Matrice des risques (Probabilité × Impact)

Risque	Prob. (1–5)	Impact (1–5)	Score	Mitigation (préventif)	Plan de contingence	Indicateur/Seuil
Charge projet sous-estimée	3	4	12	Scope strict MVP, revues bimensuelles, couper les « Should » si dérive	Déprioriser non-critiques, extension délai validée, renfort ponctuel	Burndown > 20% sur 2 sprints
Adoption faible (adhérents)	3	3	9	Mobile-first, parcours simplifié, tutoriels, tests mensuels	Campagne d'onboarding ciblée, accompagnement club pilote	< 20 utilisateurs actifs/mois ou churn > 30%
Indisponibilité HelloAsso	2	4	8	Timeouts, retries, surveillance des 5xx, dégradation contrôlée	Mode « inscription non payée » + validation manuelle ; relance paiement	Taux 5xx > 2% ou latence > 5s (p95)
Bug critique en prod	2	4	8	CI tests API, feature flags, Sentry/Logs	Rollback rapide, hotfix, correctifs priorisés	> 5 erreurs critiques/jour (Sentry)
Perte de données	1	5	5	Backups quotidiens, test de restauration mensuel, politique de rétention	Restauration à J-1, bascule instance réplica	Backup KO > 24h ou test restore échoué
RGPD / accès non autorisé	2	4	8	RBAC strict, journaux d'accès, revues périodiques, chiffrement	Déconnexion globale, rotation de clés, notification incident	Alarmes sur accès anormaux

Carte chaleur Probabilité × Impact

	Impact				
Prob.	**1**	**2**	**3**	**4**	**5**
5	5	10	15	20	25
4	4	8	12	16	20
3	3	6	9	12	15
2	2	4	6	8	10
1	1	2	3	4	5

2.6 Roadmap

La roadmap ci-dessous reprend les jalons majeurs du projet, en cohérence avec le planning général présenté au Chapitre 1. Elle distingue les versions prévues dans le cadre du diplôme (jusqu'à la V1), puis les évolutions envisagées après soutenance.

Les jalons officiels du projet sont définis dans les **GitHub Milestones**, qui servent de référence pour le pilotage, la planification et le suivi des issues. La roadmap présentée dans ce document s'aligne strictement sur ces milestones.

Milestone GitHub	Échéance	Contenu / livrables
v0.1 — Conception et cadrage (Sept–Dec 2024)	27 décembre 2025	Définition de l'architecture, du périmètre fonctionnel, diagrammes, MCD/MLD, cahier des charges, premières user stories.
v0.2 — PoC Release (Jan 2025)	17 janvier 2026	Prototype reliant front–back–API–base de données, validation technique de la stack (Next.js, Nest.js, PostgreSQL, MongoDB).
v0.3 — MVP Release (Fév 2025)	14 février 2026	Première version fonctionnelle : inscription, connexion, consultation des clubs et des informations essentielles.
v0.4 — Beta Release (Avr 2025)	25 avril 2026	Version testable par le club pilote : gestion des membres, événements, profil utilisateur, parcours mobile-first.
v1.0 — Launch (Jun 2025)	7 juin 2026	Version complète multi-clubs : rôles, paiements HelloAsso, certificats, tableau de bord simplifié, stabilisation et correctifs.

Toutes les issues de développement sont associées à ces milestones afin d'assurer une traçabilité complète et un pilotage rigoureux. Chaque milestone contient plusieurs **epics**, eux-mêmes composés de **user stories** clairement définies.

Chaque user story comporte une liste de tâches sous forme de **checkboxes** permettant de suivre l'avancement opérationnel. Pour chaque user story, une **branche Git dédiée** est créée afin d'isoler le développement.

Une fois les tâches terminées, la branche est poussée en incluant la référence de l'issue dans le message de commit, ce qui permet de **clôturer automatiquement l'issue** (#numéro).

Dans le cadre d'un projet individuel, l'usage systématique des Pull Requests (PR) serait redondant et chronophage : elles ne sont donc pas utilisées. L'objectif est de clôturer l'ensemble des user stories afin de clôturer les epics associés, puis de valider la milestone qui les regroupe.

Évolutions post-diplôme (vision produit)

— Notifications automatiques (paiement, certificat expiré)
— Tableau de bord administratif complet
— Gestion du matériel du club
— Carnet de suivi numérique
— Exports avancés (CSV / PDF)
— Montée en échelle (multi-clubs optimisé)
— Intégration FFESSM (structures départementales / régionales)
— Sécurité renforcée (2FA comité, audit avancé)

2.7 Liens utiles

— User Stories : https://www.mountaingoatsoftware.com/agile/user-stories

- MoSCoW : https://www.productplan.com/glossary/moscow-prioritization/
- PostgreSQL Docs : https://www.postgresql.org/docs/
- MongoDB Modeling : https://bit.ly/mongodb-modeling
- Architecture 3-tier : https://en.wikipedia.org/wiki/Multitier_architecture

Chapitre 3
Méthodologie et organisation

3.1 Gestion de projet avec GitHub

Gestion du projet et organisation du code

L'ensemble du projet Diving O Club est géré et suivi sur GitHub afin de garantir une traçabilité complète des décisions, du code et de l'avancement.

- **Organisation du code**
  - Dépôt GitHub unique structuré (frontend / backend / documentation)
  - Branches par fonctionnalité (`feature/...`), revues avant merge sur `develop`, puis release sur `main`
  - Messages de commit normalisés (Conventionnal Commit)
- **Planification & suivi (GitHub Projects)**
  - Tableau Kanban structuré : *Backlog* → *Next up* → *In Progress* → *In Review / Testing* → *Done*
  - Découpage du projet en *milestones* : Phase de conception et cadrage, POC, MVP, Bêta, v1.
  - Issues créées pour chaque fonctionnalité, bug ou tâche technique
- **Lien entre code et gestion**
  - Chaque issue liée à une branche
  - Pull requests associées aux issues et validées via checklist
  - Documentation des décisions techniques dans `/docs`
- **Qualité & automatisation**
  - GitHub Actions pour lint / tests / build
  - Revue systématique avant fusion sur `develop`
  - Releases taguées avec changelog versionné (`v1.0.0`, etc.)

Cette méthode assure une traçabilité complète, un développement par incrémentation contrôlé et une preuve de professionnalisation du cycle de vie logiciel.

3.2 Rituels de suivi du projet

Afin d'assurer un pilotage rigoureux du projet Diving O Club, plusieurs rituels de suivi ont été mis en place. Bien que le projet soit développé individuellement, l'organisation adoptée repose sur des **cycles courts**, inspirés du cadre agile Scrum, ce qui permet un développement itératif, incrémental et facilement ajustable selon les retours du club partenaire.

- **Sprints d'une semaine (cycle court agile)** Chaque semaine constitue un sprint avec un objectif clair (3 à 5 tâches maximum). Cette durée courte permet de livrer des incréments réguliers du produit, de limiter les risques et de faciliter la priorisation continue. Une auto-revue de sprint est réalisée en fin de cycle afin d'évaluer l'atteinte des objectifs, ajuster le backlog et préparer le sprint suivant.
- **Revue hebdomadaire (Weekly Review)** Analyse de l'avancement, mise à jour du tableau Kanban GitHub, clôture des issues terminées et identification des blocages éventuels. Ce rituel assure un recalibrage permanent de la charge, de la qualité du code et des priorités métier.

— **Tests exploratoires mensuels (avec 3 adhérents, 1 moniteur, 1 membre du comité)** Chaque mois, un test terrain est réalisé sur les parcours clés (inscription, paiement, certificat). Les retours, blocages et améliorations sont intégrés au backlog sous forme d'issues, permettant un apprentissage continu et une amélioration progressive de l'expérience utilisateur.

Ces rituels garantissent un développement structuré et agile, fondé sur des boucles courtes de feedback, une amélioration continue et une forte capacité d'adaptation aux besoins réels du club testeur.

3.2.1 User Stories et estimation de temps

User stories avec estimations :

Dans une logique inspirée des méthodes agiles, le périmètre fonctionnel du projet a été structuré en trois niveaux : **Épics**, **User Stories** et **Tâches techniques**.

— Les **Épics** regroupent les grands blocs fonctionnels du projet (ex. Authentification, Événements, Paiements).

— Chaque **User Story** exprime un besoin utilisateur clair, formulé selon le format *« En tant que [rôle], je veux [objectif], afin de [bénéfice] »*.

— Les **Tâches** découlent directement des User Stories et correspondent aux actions techniques spécifiques nécessaires à leur implémentation.

Cette structuration apporte plusieurs avantages : une meilleure lisibilité du périmètre, une priorisation facilitée, une estimation du temps plus fiable et une traçabilité directe entre besoins métier et code. L'intégralité du backlog est maintenue dans GitHub Projects pour assurer un suivi précis et incrémental.

Exemple complet : EP-01 — Authentification & Comptes

Épic EP-01 — Authentification & Comptes Assurer la création, la validation et la gestion sécurisée des comptes utilisateur, incluant la connexion, la gestion des sessions et la récupération de mot de passe.

User Stories associées

US-01 — Création de compte *En tant qu'utilisateur, je veux créer un compte sécurisé afin d'accéder à mon espace personnel.* **Critères d'acceptation :**

— Validation du format email et du mot de passe.

— Compte créé en base, rôle par défaut = `user`.

— Redirection vers l'espace personnel si inscription réussie.

US-02 — Connexion / Déconnexion *En tant qu'utilisateur, je veux me connecter avec email et mot de passe afin d'accéder à mon tableau de bord.* **Critères d'acceptation :**

— JWT d'accès court + cookie refresh httpOnly.

— Message générique en cas d'échec (anti-fuite d'information).

— Déconnexion invalide le refresh token.

US-03 — Gestion du rôle utilisateur *En tant que membre du comité, je veux modifier le rôle d'un utilisateur afin d'adapter ses permissions.* **Critères d'acceptation :**

— Mise à jour immédiate du rôle.

— Action tracée dans les logs d'audit.

US-04 — Récupération de mot de passe *En tant qu'utilisateur, je veux récupérer mon accès en cas d'oubli de mot de passe.* **Critères d'acceptation :**

— Envoi d'un lien signé avec expiration.

— Réinitialisation invalide toutes les sessions précédentes.

Tâches techniques associées

— Mise en place du module `AuthModule` dans Nest.js.
— Implémentation des DTOs (login, register, refresh, reset).
— Hashage des mots de passe avec bcrypt.
— Génération JWT (access + refresh).
— Cookies sécurisés `httpOnly`, `SameSite=Lax`.
— Middleware de validation du token.
— Route de réinitialisation du mot de passe + email signé.
— Guard RBAC pour protéger les routes sensibles.
— Ensemble de tests API : login, refresh, register, reset.

Cet exemple illustre la logique appliquée à l'ensemble des autres Épics. Le backlog complet contient plusieurs dizaines de User Stories et tâches ; afin d'éviter une surabondance dans le document, seul cet exemple est détaillé ici. Les autres éléments sont structurés et maintenus dans l'outil de gestion de projet.

Colonnes du tableau Kanban :
— **Backlog :** Fonctionnalités à développer
— **To Do :** Tâches prêtes pour le sprint
— **In Progress :** Tâches en cours (WIP limit : 3)
— **Review :** Code en attente de validation
— **Done :** Fonctionnalités livrées

3.3 Versioning GitHub et conventions

Stratégie de gestion Git — Git Flow adapté solo

Modèle retenu : Git Flow (adapté solo)

— **Branches permanentes :**
  — `main` — version stable, déployée
  — `develop` — intégration
— **Branches temporaires :**
  — `feature/<scope>` — nouvelle fonctionnalité
  — `fix/<scope>` — correctif non critique
  — `hotfix/<scope>` — correctif critique depuis `main`
  — `release/<version>`

Stratégie de merge

— Vers `develop` : pull request (relecture, CI OK), **squash & merge**
— Vers `main` : via `release/` après tests finaux
— Hotfix : branche depuis `main` -> PR vers `main` + cherry-pick vers `develop`

Nommage des branches — exemples

— `feature/auth-login`, `feature/events-calendar`, `fix/payment-status`
— `release/v1.0.0`, `hotfix/jwt-expiry`

Conventions de commit (Conventional Commits)

— `feat: auth avec JWT + refresh` *Closes #12*
— `fix: corrige statut paiement HelloAsso` *Closes #34*
— `docs: ajoute README déploiement` *Closes #7*
— `test: ajoute tests API inscriptions` *Closes #21*
— `refactor: isole service certificats` *Closes #18*
— `chore: met à jour dépendances` *Closes #3*
— `perf: optimise requête liste événements` *Closes #27*

Chaque commit référence explicitement une issue via `Closes #ID`, ce qui permet :
— la fermeture automatique de l'issue une fois la PR fusionnée ;
— une traçabilité parfaite entre le code livré et les besoins exprimés ;
— un historique clair pour le suivi du projet dans GitHub.

Pull Requests (PR)

— Template PR incluant checklist :
  — Issue liée (*Closes #12*)
  — Tests ajoutés / mis à jour
  — Lint & CI OK
  — Impact sécurité (auth / rôles) vérifié
— Règles : pas de commit direct sur `main`, 1 PR = 1 sujet, captures UI si besoin

Protection des branches

— `main` : protégée (CI obligatoire, squash & merge uniquement, review requise)
— `develop` : CI obligatoire, pas de push direct en production

Versioning & releases

— SemVer `MAJOR.MINOR.PATCH`
— `v1.0.0` = MVP (juin 2026)
— `v1.1.0` = dashboard + notifications
— `v1.2.0` = matériel + carnet de plongée
— Changelog généré depuis commits (`feat`/`fix`) — `CHANGELOG.md`
— Tags sur `main` lors des releases

Traçabilité GitHub

— Chaque US = issue (labels : feature, bug, security, docs, P1–P3)
— Branche issue-based : `feature/123-events-calendar`
— Commits & PR référencent l'issue (`fixes #123`)
— Milestones alignées avec roadmap (MVP, v1.1, v1.2)

CI/CD (qualité)

— GitHub Actions : `lint -> test -> build` sur PR ; déploiement sur tag `v*`
— Tests sur endpoints critiques (auth, inscriptions, paiements, certificats)
— Scan basique de vulnérabilités

Gestion des conflits & imprévus

— Résolution locale puis rebase avant PR
— Urgences via `hotfix/`, patch immédiat, sync vers `develop`

Documentation

— `README.md`
— User stories

3.4 Planification et outils de suivi

Planification et pilotage du projet

La planification du projet repose sur une approche duale :

— **Roadmap GitHub** pour la vision macro (jalons, dépendances, versions)
— **Kanban GitHub Projects** pour le suivi opérationnel au quotidien (User Stories, issues, flux de production)

Cette organisation permet d'allier **vision stratégique long terme** et **exécution incrémentale contrôlée**.

Roadmap GitHub — Vision produit

— Découpage par versions :
  — `MVP` = avril 2026
  — `v1.0` = rentrée clubs (fin août 2026)
  — `v1.1` = dashboard + notifications (sept. 2026)
  — `v1.2` = matériel + carnet de plongée (déc. 2026)
— **Milestones** rattachés à chaque version (issues associées)
— Dépendances visibles : backend -> frontend -> intégration -> déploiement
— Gestion des risques : intégration HelloAsso, sécurité, performance
— Communication claire avec les parties prenantes (club testeur)

Kanban GitHub Projects — Suivi opérationnel

— Colonnes : *Backlog -> To Do -> In Progress -> Review -> Done*
— Limite WIP : 2–3 tâches maximum simultanément (évite la dispersion)
— 1 issue = 1 User Story (estimée, assignée, reliée à un milestone)
— Automatisations :
  — Passage automatique de l'issue lorsque la PR est fusionnée
  — Fermeture automatique via `Closes #XX`
— PR obligatoires pour fusionner sur `develop` ou `main`
— Indicateurs suivis : lead time, throughput, cycle time

3.5 Estimation de temps et planification

Estimation du temps et faisabilité

L'estimation du temps est basée sur le périmètre MVP (v1.0) composé des dix User Stories essentielles. Le développement est planifié en alternance ($\approx$ 3 jours / semaine en entreprise), soit environ $\approx$ 12 jours-homme par mois.

Estimation par fonctionnalité (MVP)

Bloc fonctionnel	**Estimation**
Authentification & rôles (US-01 / US-02)	3,5 jours
Calendrier & gestion d'événements (US-03 / US-04)	5 jours
Inscriptions & annulations (US-05 / US-06)	3,5 jours
Paiements HelloAsso (US-07 / US-08)	4,5 jours
Certificats médicaux (US-09 / US-10)	3,5 jours
Total MVP	**20–22 jours**

En rythme alternance : $\approx$ 2 mois calendrier (3 jours / semaine). Ce planning s'inscrit dans la phase prévue : **Décembre -> Avril — Développement back-end / front-end**.

Estimation par phase projet

Phase	**Période**	**Objectif**	**Charge**
Conception & cadrage	Oct–Déc 2025	UML, MCD/MLD, cahier des charges, maquettes	$\sim$ 10 j
POC (stack)	Déc 2025	Validation Next.js / Nest.js / PostgreSQL + MongoDB	$\sim$ 3 j
Dév. back-end (MVP)	Jan–Mar 2026	API, Auth, Events, Inscriptions, Certificats, Paiements (HelloAsso)	$\sim$ 22 j
MVP	**Avr 2026**	**Version utilisable en conditions réelles (club)**	–
Front + intégrations	Mai–Juin 2026	UI Next.js/React, intégration, tests, perf	$\sim$ 15 j
Stabilisation & bêta	Juil 2026	Corrections, retours club pilote (Aquaclub21)	$\sim$ 8 j
V1 (rentrée clubs)	**Fin août 2026**	Déploiement final & documentation	–

Ce volume est cohérent avec la durée d'alternance (9 mois effectifs).

Validation de la faisabilité

— Le périmètre MVP a été réduit au module cœur pour rester réalisable.
— Les fonctionnalités secondaires sont planifiées en v1.1 / v1.2 (après retours terrain).
— Le planning est aligné sur la roadmap produit et les deadlines CDA.

3.6 Liens utiles

— GitHub Flow/PRs : https://docs.github.com/pull-requests

— Git Flow : https://bit.ly/gitflow-atlassian
— GitHub Projects : https://bit.ly/github-projects
— GitHub Roadmap : https://bit.ly/github-roadmap
— GitHub Milestones : https://bit.ly/github-milestones
— User Stories : https://www.mountaingoatsoftware.com/agile/user-stories
— Estimation de temps : https://bit.ly/time-estimation

Chapitre 4

Conception fonctionnelle et technique

Approche de conception :

Dans le cadre du projet *Diving O Club*, j'ai adopté une démarche de conception itérative et guidée par les besoins métier réels des clubs de plongée. La conception a été menée **avant tout développement**, en lien étroit avec le club pilote (Aquaclub 21) afin de sécuriser les choix fonctionnels et techniques et de limiter au maximum les refactorisations ultérieures.

Je suis parti d'un recueil des besoins sous forme d'ateliers, d'entretiens informels et de retours d'expérience des membres du comité directeur, des encadrants et des adhérents. Ces échanges ont permis de formaliser une première vision produit, puis de la traduire en **épics** et **user stories** priorisées (méthode MoSCoW) dans un backlog structuré. Chaque user story est associée à des critères d'acceptation clairs (méthode Gherkin), ce qui assure une traçabilité directe entre le besoin métier, la conception et les futurs tests.

À partir de ce socle fonctionnel, j'ai structuré la conception en plusieurs axes complémentaires :

— **Conception fonctionnelle** : modélisation des acteurs et des cas d'usage (UML), description des parcours utilisateurs clés (inscription, gestion des certificats, inscriptions aux événements, suivi des paiements, etc.) et identification des flux alternatifs et cas d'erreur.

— **Conception des interfaces** : élaboration de zonings mobile first, puis de wireframes basse fidélité et de maquettes haute fidélité alignées sur une charte graphique cohérente avec l'univers de la plongée. Ces éléments servent de base au découpage en composants front-end réutilisables.

— **Conception des données** : modélisation du **MCD**, puis du **MLD** et du **MPD** (méthode Merise) pour la partie relationnelle (PostgreSQL), en complément d'un schéma documentaire pour les journaux et traces techniques (MongoDB).

— **Conception technique** : définition d'une architecture 3 tiers (présentation, logique métier, données), choix des technologies (React/Next.js, API Node.js/Nest.js, PostgreSQL, MongoDB), et formalisation des principaux flux via des diagrammes de séquence.

Le **processus de validation** repose sur plusieurs boucles courtes :

— Relecture et ajustement des cas d'usage et des maquettes avec les représentants métier (comité directeur, encadrants).

— Vérification de la cohérence des modèles de données et des diagrammes de séquence avec les user stories et les scénarios d'utilisation.

— Alignement des choix techniques avec les contraintes du projet (sécurité, volumétrie, multi-club à terme, maintenabilité) et validation avec mes encadrants pédagogiques.

Ce travail de conception, documenté et versionné (diagrammes, maquettes, modèles de données, backlog), sert ainsi de référence pour le développement, les tests et la maintenance de *Diving O Club*. Les sections suivantes détaillent chacune de ces dimensions (Use Cases, interfaces, base de données, architecture 3 tiers) et montrent comment elles s'articulent pour former un ensemble cohérent.

4.1 Use Cases et diagrammes UML

Les Use Cases modélisent les interactions entre les acteurs et le système pour identifier les fonctionnalités essentielles. Cette approche centrée utilisateur garantit que l'application répond aux besoins réels des clubs de plongée et des différents profils utilisateurs (adhérents, moniteurs, membres du comité directeur, administrateur de plateforme).

Même si je suis l'unique développeur du projet, les diagrammes UML jouent un rôle structurant dans la conception. Ils facilitent la communication avec les parties prenantes du club pilote (Aquaclub21), notamment lors des ateliers de validation fonctionnelle. Ils permettent également d'éviter les ambiguïtés, de clarifier les périmètres fonctionnels et de sécuriser les décisions avant le développement.

Le diagramme global des cas d'usage présenté ci-dessous regroupe l'ensemble des interactions possibles selon les rôles utilisateurs. Il met également en évidence l'intégration du service externe HelloAsso pour la gestion des paiements et du statut des transactions.

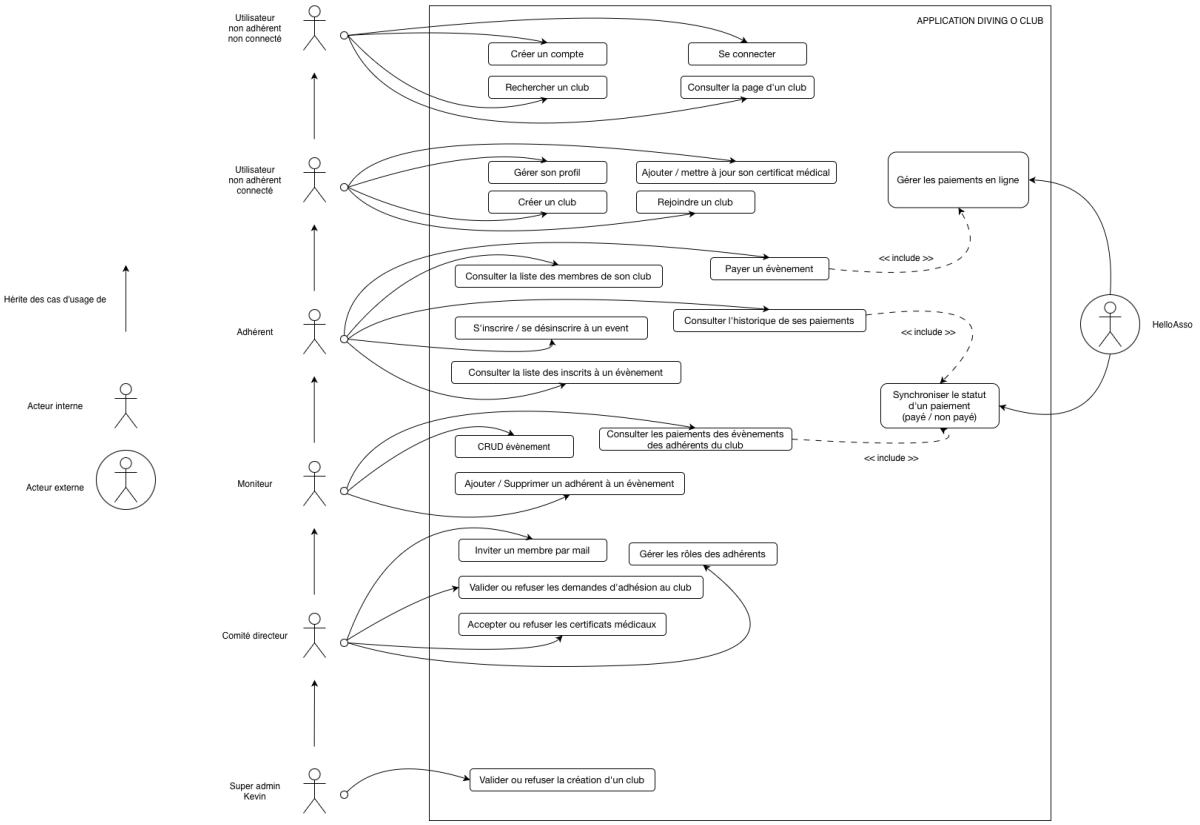


Figure 4.1 – Diagramme global des cas d'usage de l'application Diving O Club

La modélisation des cas d'usage a permis d'identifier :

- les fonctionnalités accessibles selon le statut utilisateur (non connecté, connecté sans club, adhérent, moniteur, comité directeur, super administrateur),
- les héritages fonctionnels progressifs selon les rôles,
- les interactions critiques nécessitant une validation métier (adhésion, certificats médicaux, gestion des inscriptions),
- les dépendances externes (HelloAsso),
- les cas alternatifs et erreurs à gérer (refus d'adhésion, capacité maximale atteinte, paiement échoué).

Cette analyse structurante a servi de base :

- à la définition du backlog et des User Stories,
- à la priorisation des versions (MVP, Bêta, V1),
- à la conception des APIs et des règles de gestion,
- aux scénarios de tests d'acceptation.

4.2 Diagrammes de séquence

Les diagrammes de séquence détaillent les interactions temporelles entre les différents composants du système pour chaque cas d'usage. Ils constituent un support essentiel pour clarifier la circulation des données, les responsabilités techniques et les enchaînements conditionnels avant la phase de développement.

Dans le cadre de l'application Diving O Club, ces diagrammes permettent de structurer l'architecture autour des trois couches du projet : le frontend (Next.js), la couche API (Nest.js) et la couche données (PostgreSQL et services externes tels que HelloAsso). Ils servent également de référence pour la rédaction des tests d'intégration et pour la validation des exigences fonctionnelles auprès des responsables du club pilote.

Diagramme de séquence : Authentification utilisateur

Le premier diagramme présente le processus complet de connexion d'un utilisateur, depuis la saisie des identifiants jusqu'à l'obtention du jeton JWT et la redirection selon la situation de l'utilisateur (rattaché ou non à un club). Ce diagramme met en évidence :

— la vérification des identifiants en base de données,
— la génération sécurisée du token JWT,
— la gestion des erreurs d'authentification,
— la récupération du club associé à l'utilisateur,
— la logique conditionnelle de redirection.

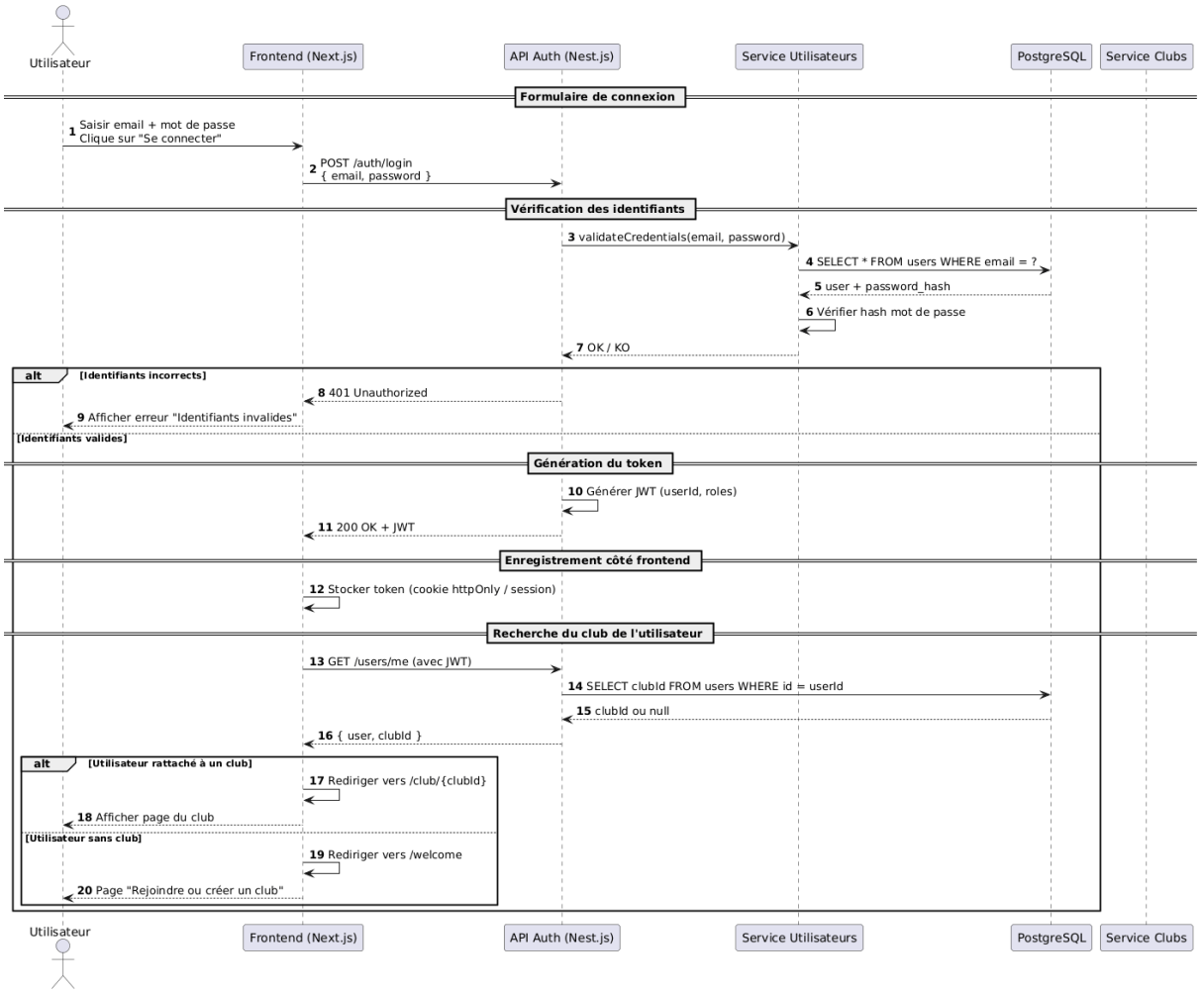


Figure 4.2 – Diagramme de séquence – Processus de connexion utilisateur

Diagramme de séquence : Inscription à un événement et paiement

Le second diagramme décrit le flux complet d'inscription à un événement, en distinguant les deux scénarios possibles : événement gratuit et événement payant. Il illustre notamment :

— la vérification du nombre de places disponibles,
— la création de l'inscription en base,
— la détection du caractère payant de l'événement,
— la redirection vers HelloAsso,
— le traitement du webhook de confirmation,
— la mise à jour du statut de participation.

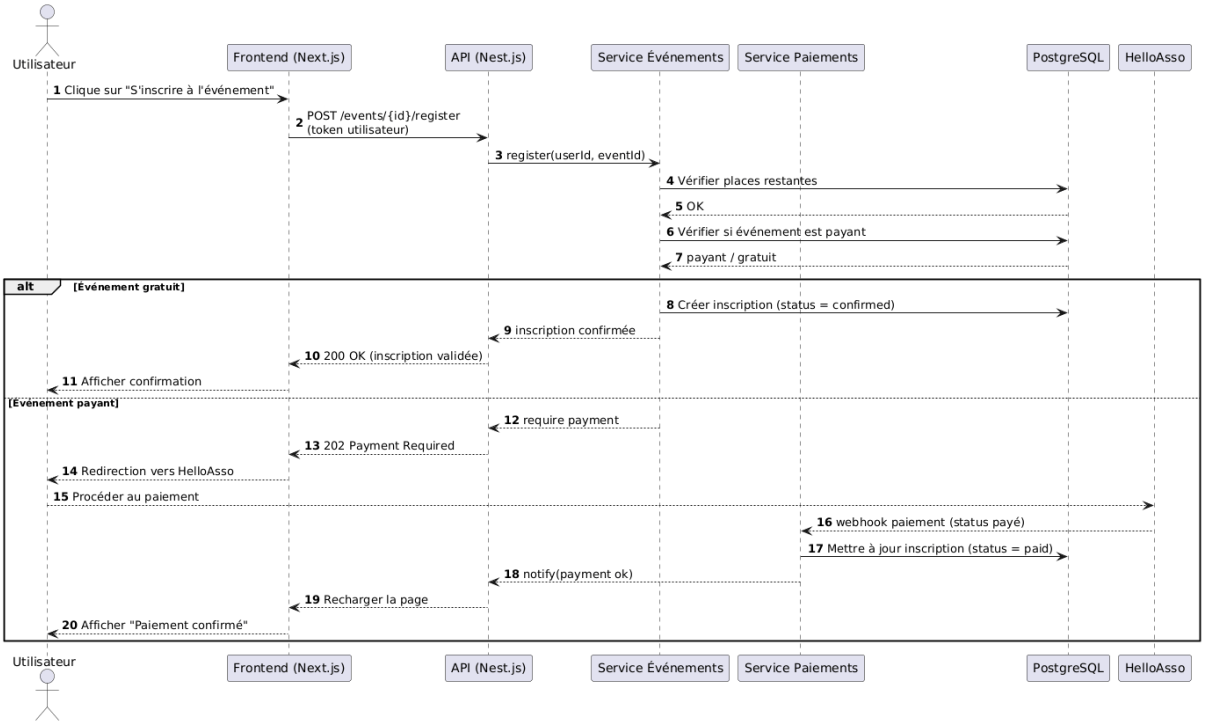


Figure 4.3 – Diagramme de séquence – Inscription à un événement avec gestion du paiement

Les diagrammes de séquence ont permis d'identifier les points sensibles du système, notamment la synchronisation des paiements, la gestion des erreurs réseau, la cohérence transactionnelle et la nécessité de sécuriser les échanges par jetons signés. Ils constituent une référence directe pour l'implémentation des services backend et pour la définition des contrats API.

4.3 Conception de l'interface graphique

La conception de l'interface graphique de l'application Diving O Club s'inscrit dans une approche **Mobile First**, en cohérence avec les usages réels des adhérents de clubs de plongée, dont la majorité consulte leurs informations depuis un smartphone. À ce stade du projet, seules les interfaces mobiles ont été conçues afin de garantir une expérience fluide, rapide et intuitive pour les utilisateurs prioritaires : adhérents, encadrants et membres du comité.

Cette démarche progressive permet de structurer l'ergonomie avant d'étendre la conception aux formats tablettes et desktop. La simplicité, la lisibilité et la réduction du nombre d'actions nécessaires guident l'ensemble des choix UX afin de favoriser l'adoption et limiter la charge cognitive. Les wireframes et zonings servent de base aux maquettes haute fidélité et facilitent la mise en œuvre front-end dans Next.js.

4.3.1 Zoning

Le zoning constitue la première étape de structuration visuelle des interfaces. Il permet de définir l'organisation spatiale des éléments principaux sans se concentrer sur l'aspect graphique final. Cette étape clarifie la hiérarchie de l'information, les zones d'interaction, les points d'attention de l'utilisateur et les parcours fonctionnels.

Elle sert de support pour valider :

— la disposition des blocs fonctionnels,

— la cohérence des parcours utilisateurs,

— la lisibilité des actions principales,

— la séparation entre navigation, contenu et éléments contextuels.

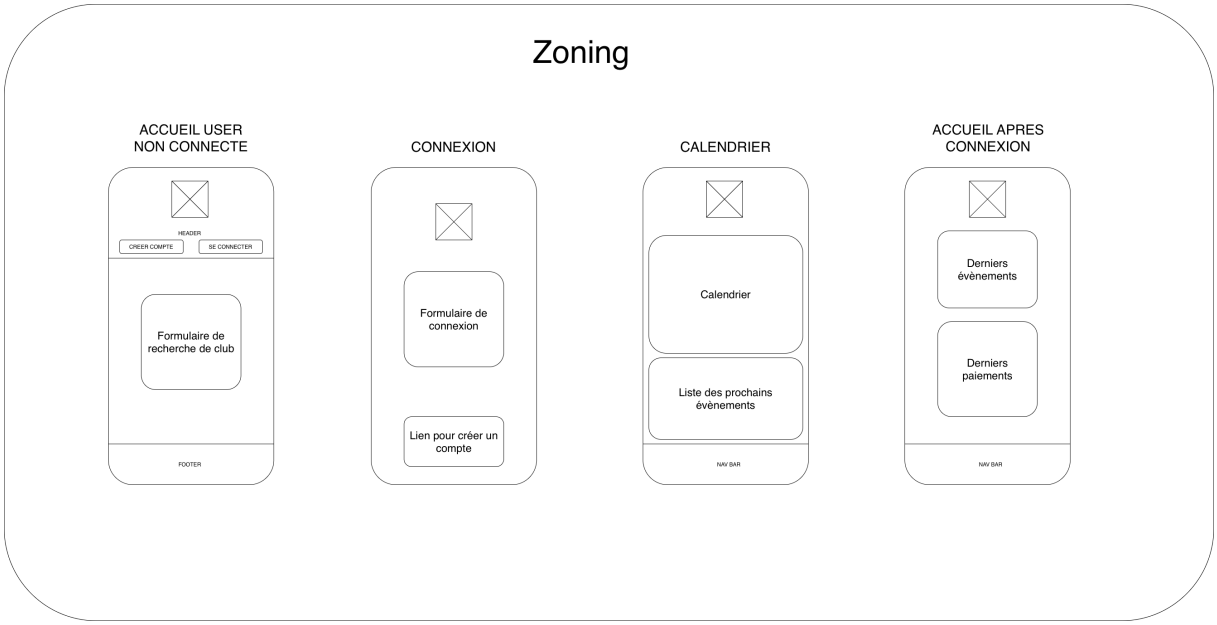


Figure 4.4 – Zoning Mobile First – organisation des pages principales

Le zoning réalisé met en avant une structure simple et intuitive : une zone supérieure dédiée à l'identité visuelle du club ou de l'application, une zone centrale contenant l'élément fonctionnel principal de chaque écran (recherche de clubs, connexion, calendrier, informations du tableau de bord), et une zone inférieure réservée à la navigation persistante. Ce choix permet à l'utilisateur de se repérer facilement, de réaliser les actions essentielles en un minimum d'interactions et de limiter la charge visuelle.

4.3.2 Wireframes

Les wireframes représentent la seconde étape de la conception d'interface après le zoning. Ils permettent de détailler la structure fonctionnelle des écrans en intégrant les premiers éléments interactifs, les libellés, les champs de formulaire et la logique de navigation. Contrairement aux maquettes, les wireframes restent volontairement neutres visuellement afin de se concentrer sur l'expérience utilisateur et les parcours fonctionnels.

Cette étape est essentielle pour :

— valider les parcours utilisateurs avant développement,
— préciser les interactions clés (connexion, inscription, navigation),
— identifier les contraintes ergonomiques sur mobile,
— limiter le risque de révisions tardives et coûteuses,
— servir de référence au développement front-end dans Next.js.

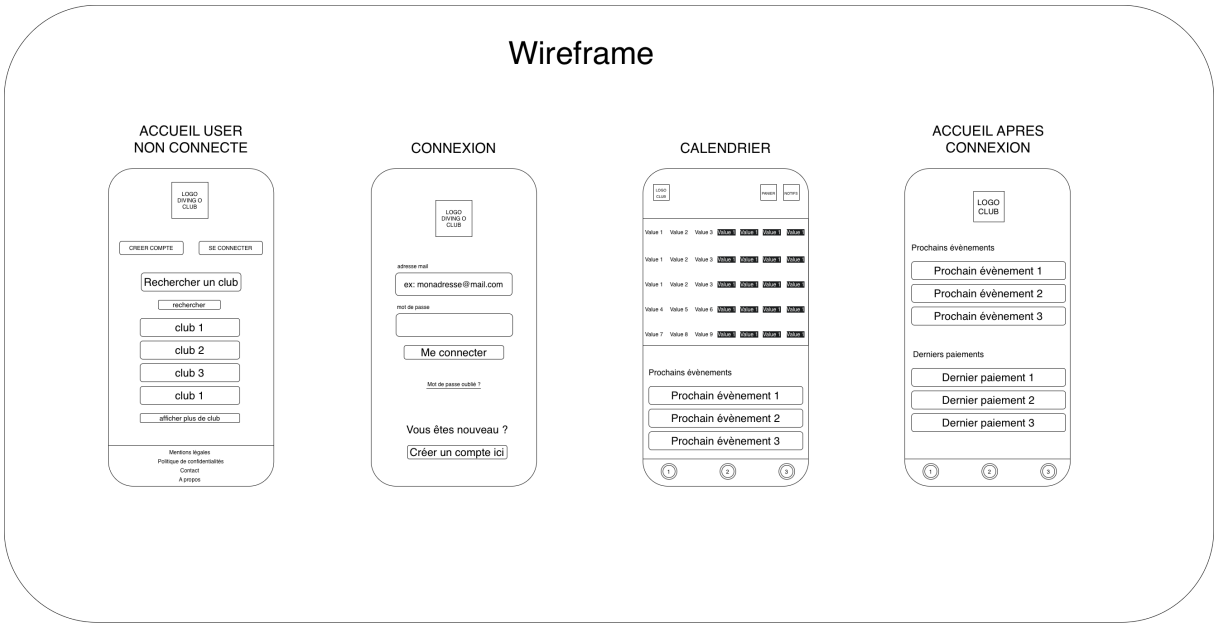


Figure 4.5 – Wireframes Mobile First – écrans principaux de l'application

Les wireframes définis pour Diving O Club se concentrent sur les écrans prioritaires du MVP :

— **Accueil non connecté** : recherche de club et accès à l'authentification,
— **Connexion** : formulaire simple, clair, avec accès à la création de compte,
— **Calendrier des évènements** : visualisation des prochains évènements du club,
— **Accueil après connexion** : accès aux évènements et aux derniers paiements.

Les choix UX mettent l'accent sur :

— une navigation inférieure persistante pour un repérage immédiat,
— des actions principales accessibles en un seul clic,
— une information hiérarchisée pour limiter la charge cognitive,
— des composants réutilisables facilitant l'implémentation et la cohérence visuelle.

Cette étape a permis de sécuriser les parcours essentiels avant la réalisation des maquettes haute fidélité.

4.3.3 Maquettage

Les maquettes haute fidélité viendront définir l'identité visuelle finale de l'application. Elles s'appuieront sur une charte graphique inspirée de l'univers de la plongée sous-marine, avec une dominante de couleurs bleu, blanc et noir. Le bleu évoque l'environnement aquatique, la profondeur et la sérénité ; le blanc apporte de la clarté et renforce la lisibilité ; le noir permet de structurer et de créer un contraste élégant.

L'esthétique générale intégrera des éléments visuels rappelant l'exploration sous-marine, tels que des dégradés évoquant les variations de lumière sous l'eau, des illustrations stylisées de bulles d'air, de poissons ou de végétation marine. L'objectif est de proposer une interface moderne et immersive, tout en restant sobre et fonctionnelle.

L'interface adoptera également une approche inclusive à destination des utilisateurs peu familiers avec les outils numériques : le nombre de boutons sera volontairement limité, chaque action sera clairement identifiable, et les libellés seront explicites et non techniques. Ces choix visent à réduire la charge cognitive, éviter la confusion et permettre une prise en main immédiate, y compris pour les adhérents non technophiles.

Cette approche permet d'assurer une cohérence visuelle sur l'ensemble des écrans, de renforcer l'identité du projet et d'améliorer l'expérience utilisateur en créant un environnement graphique harmonieux et identifiable.

4.3.4 Outils de conception et diagrammes

Pour la réalisation des différents schémas et supports visuels nécessaires à la conception du projet, plusieurs outils complémentaires ont été utilisés afin d'obtenir un rendu professionnel, structuré et adapté aux besoins techniques.

Les diagrammes de séquence ont été créés avec **PlantUML**, un outil basé sur la génération de diagrammes à partir de code textuel. Ce choix permet une grande précision dans la structuration des interactions, une meilleure maintenabilité et la possibilité de modifier ou itérer rapidement sans devoir reconstruire le schéma visuellement. Cette approche s'est révélée particulièrement efficace pour des séquences complexes impliquant plusieurs acteurs, services et couches applicatives.

L'ensemble des autres diagrammes — including les Use Cases, le MCD, le MLD, le MPD, l'architecture 3 tiers et les schémas organisationnels — ont été réalisés avec **app.diagrams**. Cet outil offre une interface graphique intuitive, des gabarits adaptés aux standards UML, des options de connecteurs intelligents et des fonctionnalités facilitant l'alignement, la lisibilité et la mise en page. Ces capacités sont particulièrement utiles pour produire des schémas visuels nets, hiérarchisés et facilement exploitables dans la documentation.

L'usage combiné de ces deux outils permet :

— une production rapide et cohérente des diagrammes,

— une meilleure lisibilité pour les parties prenantes non techniques,

— une mise à jour simplifiée des schémas lors de l'évolution du projet,

— une compatibilité directe avec l'exportation en formats PDF et PNG pour intégration au dossier,

— une homogénéité visuelle dans l'ensemble de la documentation.

Cette organisation garantit une documentation claire, professionnelle et alignée avec les attentes du jury, tout en favorisant une communication efficace autour du projet.

4.3.5 Charte graphique

Dans cette sous-section, vous devez détailler votre charte graphique complète. Le jury attend une cohérence visuelle et une identité forte.

Couleurs

Votre palette de couleurs : *[Définissez votre palette avec les codes hexadécimaux]*

Typographie

Votre système typographique : *[Définissez vos polices et leurs usages]*

Logo

Votre logo et son utilisation : *[Décrivez votre logo et ses variantes]*

4.4 Conception de base de données

La conception de la base de données s'appuie sur la méthode Merise, en suivant une progression structurée du Modèle Conceptuel de Données (MCD) vers le Modèle Logique de Données (MLD), puis vers le Modèle Physique de Données (MPD). Cette démarche permet de garantir la cohérence du modèle, la conformité avec les besoins fonctionnels et l'optimisation

du stockage des informations essentielles à la gestion des clubs de plongée, des adhérents, des événements et des paiements.

L'architecture retenue repose sur une double logique de stockage : **PostgreSQL** pour la gestion des données transactionnelles (adhésions, rôles, certificats, inscriptions aux événements, paiements) avec intégrité forte, et **MongoDB** pour le stockage des journaux d'activité, rapports et suivis applicatifs nécessitant une structure flexible et évolutive. Cette répartition permet d'optimiser la performance selon la nature des données, tout en garantissant la fiabilité, la traçabilité et l'évolutivité du système.

Les contraintes d'intégrité, les clés primaires, les clés étrangères, ainsi que les index sont définis pour assurer la robustesse du modèle et optimiser les performances sur les opérations les plus fréquentes. L'approche retenue permet également d'accompagner l'évolution progressive du projet au fil des versions (MVP, Bêta, V1) sans restructuration majeure du socle applicatif.

4.4.1 MCD (Modèle Conceptuel de Données)

Le Modèle Conceptuel de Données (MCD) représente les entités principales du système Diving O Club ainsi que leurs relations. Il permet de structurer les informations nécessaires au fonctionnement de la plateforme : gestion des utilisateurs, clubs, adhésions, événements, inscriptions et paiements. Ce modèle garantit que les besoins fonctionnels exprimés dans les cas d'usage se traduisent en données correctement organisées, cohérentes et exploitables pour les versions MVP, Bêta et V1 du projet.

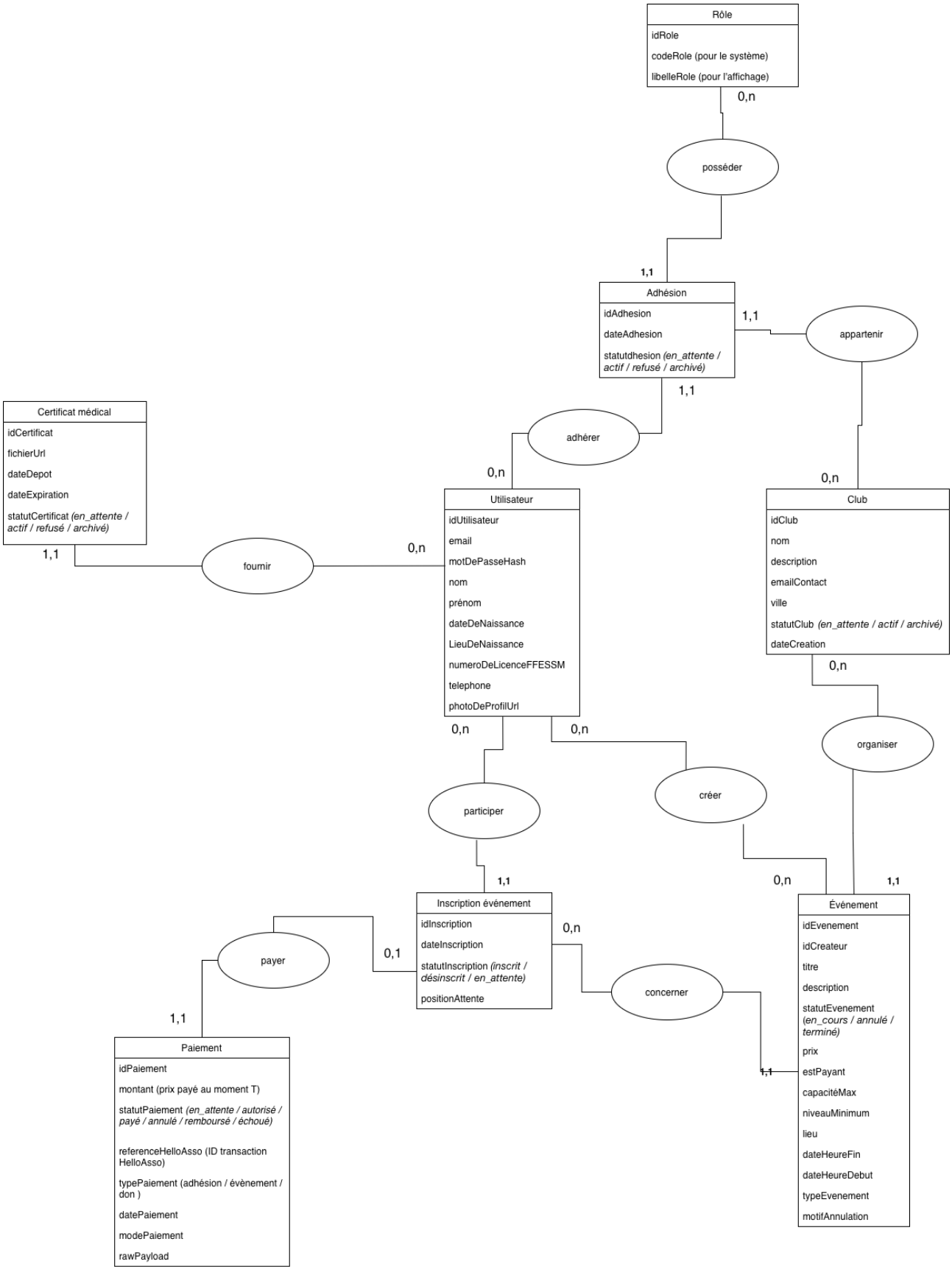


Figure 4.6 – Modèle Conceptuel de Données de l'application Diving O Club

Principales entités

- **Utilisateur** : représente l'ensemble des membres de la plateforme (visiteurs inscrits, adhérents, moniteurs, membres du comité directeur). Contient les informations d'identité, de contact et de profil.
- **Club** : correspond aux structures associatives affiliées FFESSM pouvant accueillir des adhérents et organiser des événements.
- **Adhésion** : associe un utilisateur à un club, avec un rôle (adhérent, moniteur, comité directeur) et un statut (en attente, validée, refusée).
- **Certificat Médical** : document attaché à un utilisateur, permettant de contrôler la conformité à la réglementation FFESSM.
- **Événement** : sorties plongée, entraînements piscine, voyages… organisés par un club.
- **Inscription Événement** : lie un utilisateur à un événement, avec présence, annulation ou liste d'attente.
- **Paiement** : stocke le statut de paiement d'un utilisateur pour un événement, synchronisé avec HelloAsso.

Relations principales

- Un **Utilisateur** peut appartenir à plusieurs **Clubs** via les **Adhésions**.
- Un **Club** peut organiser plusieurs **Événements**.
- Un **Utilisateur** peut s'inscrire à plusieurs **Événements**.
- Un **Certificat Médical** appartient à un seul **Utilisateur**.
- Un **Paiement** est lié à une **Inscription Événement**.

Contraintes d'intégrité

- **Un utilisateur ne peut avoir qu'un seul certificat médical actif** : garantit la conformité légale et la cohérence des données.
- **Un paiement est obligatoirement rattaché à une inscription à un événement** : empêche tout paiement isolé ou impossible à associer.

4.4.2 MLD (Modèle Logique de Données)

Le Modèle Logique de Données (MLD) traduit le MCD en un ensemble de tables relationnelles adaptées à PostgreSQL. Il formalise les clés primaires, les clés étrangères et les cardinalités entre les entités, en respectant les règles métier identifiées pour Diving O Club : gestion des utilisateurs, des clubs, des adhésions, des événements, des inscriptions et des paiements.

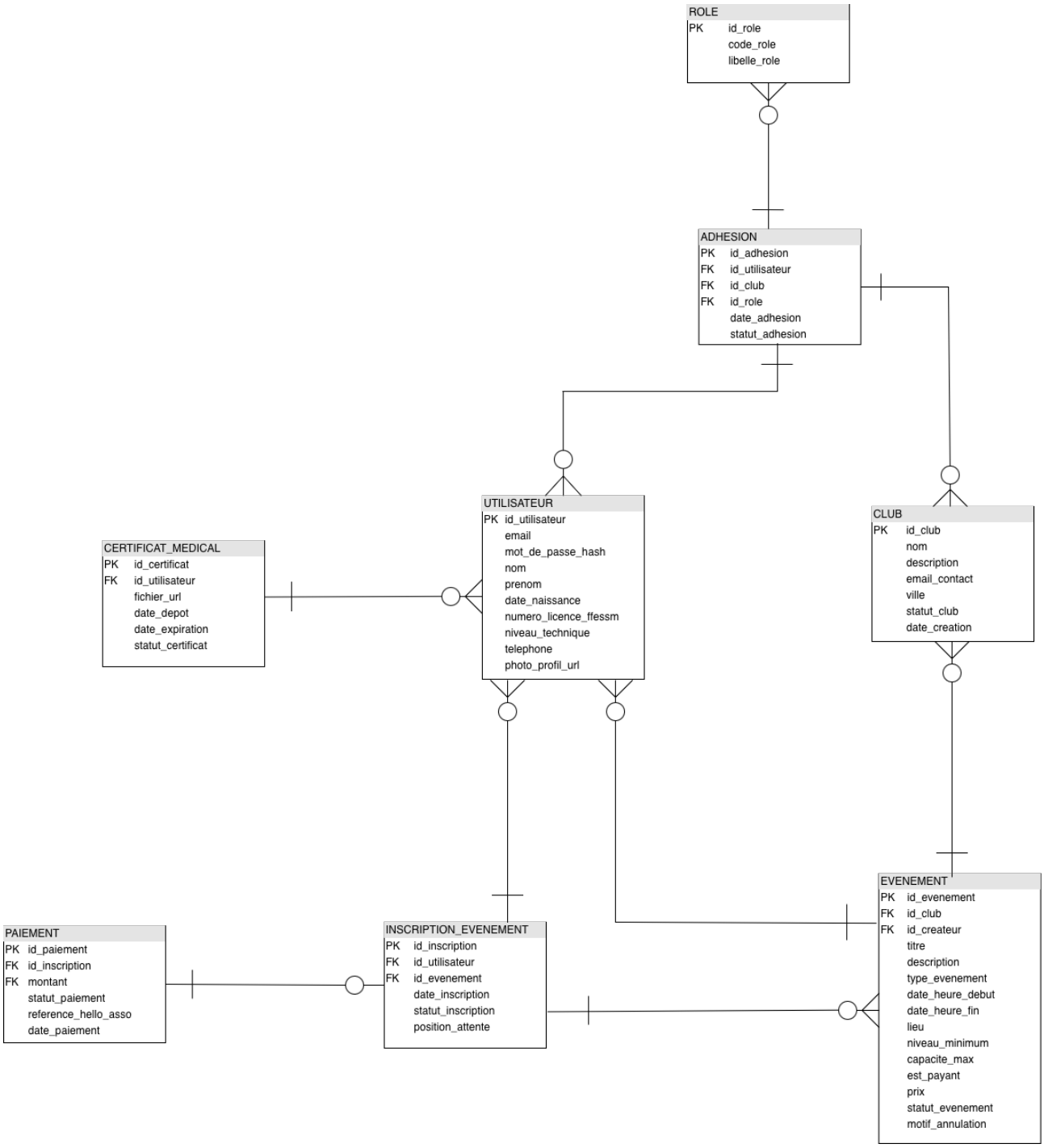


Figure 4.7 – Modèle Logique de Données de l'application Diving O Club

Le MLD est structuré autour des tables principales suivantes :

— **UTILISATEUR** : table centrale pour l'authentification et le profil des membres de la plateforme (adhérents, moniteurs, membres du comité directeur, etc.).

— **CLUB** : représente les clubs de plongée pouvant accueillir des adhérents et organiser des événements.

— **ROLE** : liste les rôles possibles au sein d'un club (adhérent, moniteur, comité directeur) afin de ne pas les dupliquer dans plusieurs tables.

— **ADHESION** : table d'association entre `UTILISATEUR`, `CLUB` et `ROLE`, permettant de gérer l'ap-

partenance d'un utilisateur à un club ainsi que son rôle et le statut de l'adhésion.

— **CERTIFICAT_MEDICAL** : stocke les certificats médicaux associés aux utilisateurs, avec leur statut (en attente, validé, refusé) et leurs dates de validité.
— **EVENEMENT** : regroupe les événements créés par un club (sorties mer, entraînements piscine, etc.), avec les informations de lieu, horaires, niveau requis et capacité.
— **INSCRIPTION_EVENEMENT** : table d'association entre `UTILISATEUR` et `EVENEMENT`, permettant de gérer les inscriptions, désinscriptions et listes d'attente.
— **PAIEMENT** : contient les informations de paiement liées aux inscriptions, y compris le montant, le statut et la référence HelloAsso.

Les relations entre les tables, représentées en notation « crow's foot » dans le schéma, garantissent notamment que :

— un **UTILISATEUR** peut avoir plusieurs **ADHESIONS** mais une adhésion ne concerne qu'un seul utilisateur et un seul club ;
— un **CLUB** peut organiser plusieurs **EVENEMENTS** ;
— un **UTILISATEUR** peut avoir plusieurs **CERTIFICAT_MEDICAL** au cours du temps (historique) ;
— un **EVENEMENT** peut avoir plusieurs **INSCRIPTION_EVENEMENT**, chacune liée à un utilisateur ;
— un **PAIEMENT** est toujours rattaché à une **INSCRIPTION_EVENEMENT**, ce qui évite les paiements « orphelins ».

Ce modèle logique sert de base directe à la construction du Modèle Physique de Données (MPD), dans lequel chaque table est enrichie avec les types de données PostgreSQL, les contraintes d'unicité, les clés étrangères et les index nécessaires aux performances.

4.4.3 MPD (Modèle Physique de Données)

Le Modèle Physique de Données (MPD) constitue la traduction du MLD en structures concrètes adaptées au SGBD retenu pour l'application : PostgreSQL. Il définit les types de données, les clés primaires, les clés étrangères, les contraintes d'unicité ainsi que les index nécessaires pour optimiser les performances des opérations les plus courantes (recherche de clubs, gestion des adhésions, inscriptions aux événements et suivi des paiements).

Les tables sont typées selon les besoins fonctionnels, avec notamment :

— **UUID** pour les identifiants (`id_utilisateur`, `id_club`, `id_evenement`) afin de permettre l'évolutivité multi-tenant et la sécurité ;
— **VARCHAR** pour les champs texte structurés (nom, email, intitulés) ;
— **BOOLEAN** pour les statuts logiques (validation, présence, désactivation) ;
— **DATE** ou **TIMESTAMP** pour les éléments temporels ;
— **ENUM** ou **CHECK** pour les statuts métiers (adhésion, certificat, paiement).

Les clés primaires (PK) garantissent l'unicité de chaque enregistrement, tandis que les clés étrangères (FK) assurent la cohérence des relations entre :

— utilisateur et adhésion ;
— club et événement ;
— événement et inscription ;
— inscription et paiement ;
— utilisateur et certificat médical.

Des index sont ajoutés afin d'améliorer les performances sur les opérations récurrentes, notamment :

— recherche par club (`idx_utilisateur_club`) ;
— accès aux événements futurs (`idx_evenement_date`) ;
— statut des adhésions et certificats (`idx_adhesion_statut`, `idx_certificat_statut`) ;
— synchronisation HelloAsso (`idx_paiement_statut`).

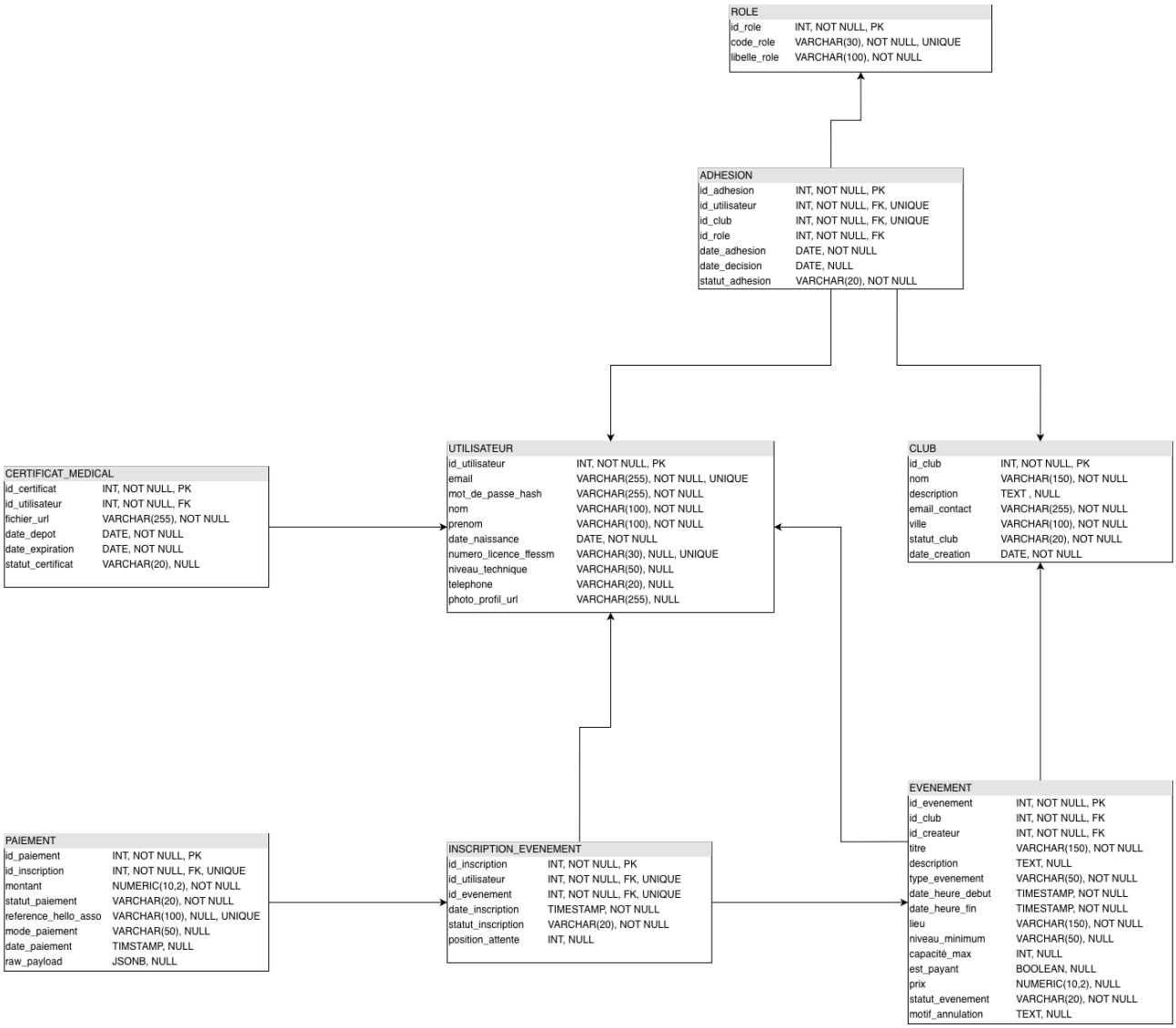


Figure 4.8 – Modèle Physique de Données de l'application Diving O Club

Contraintes d'intégrité physiques

— **Un utilisateur ne peut pas s'inscrire deux fois au même événement**
(contrainte d'unicité sur `(id_utilisateur, id_evenement)` dans `INSCRIPTION_EVENEMENT`)

— **Un paiement ne peut exister sans inscription associée**
(FK obligatoire `id_inscription_evenement` dans `PAIEMENT`)

Le MPD permet ainsi de garantir la fiabilité des données, la conformité réglementaire FFESSM sur les certificats médicaux, ainsi qu'une base solide pour l'évolution du système vers la gestion multi-clubs et l'intégration avancée HelloAsso prévue en V1.

Exemple de contraintes PostgreSQL :

Contraintes d'intégrité dans la base de données

Dans le cadre de la conception du Modèle Physique de Données (MPD), plusieurs contraintes d'intégrité ont été ajoutées afin de garantir la cohérence et la fiabilité des informations stockées dans la base PostgreSQL. Ci-dessous sont présentées deux des contraintes principales mises en place.

1. Contrainte d'unicité sur l'email utilisateur

Cette contrainte permet de garantir qu'aucun utilisateur ne puisse partager la même adresse email. Elle empêche la création de doublons et assure une identification unique lors de l'authentification.

```
ALTER TABLE utilisateur
  ADD CONSTRAINT uq_utilisateur_email UNIQUE(email);
```

2. Intégrité référentielle entre les inscriptions et les événements

Cette contrainte garantit qu'une inscription ne peut exister que si l'événement associé est présent dans le système. L'option `ON DELETE CASCADE` assure qu'en cas de suppression d'un événement, les inscriptions liées seront automatiquement supprimées, évitant ainsi des données orphelines.

```
ALTER TABLE inscription_evenement
  ADD CONSTRAINT fk_inscription_evenement
    FOREIGN KEY (id_evenement) REFERENCES evenement(id_evenement)
    ON DELETE CASCADE;
```

Exemple de document MongoDB (logs et audit)

En complément de la base de données relationnelle PostgreSQL, MongoDB est utilisé pour stocker des journaux techniques (logs) et des informations d'audit sous forme de documents JSON. Cette approche permet de conserver un historique détaillé des actions des utilisateurs sans alourdir le schéma transactionnel principal.
L'exemple ci-dessous illustre un document MongoDB représentant l'enregistrement d'une inscription à un événement :

```
{
  "_id": ObjectId("..."),
  "userId": "u_123",
  "clubId": "c_456",
  "action": "event_registration_created",
  "timestamp": ISODate("2026-03-15T19:30:00Z"),
  "metadata": {
    "evenementId": "e_789",
    "inscriptionId": "i_1011",
    "statutInscription": "inscrit",
    "ipAddress": "192.168.1.42",
    "userAgent": "Mozilla/5.0 (Mobile; iOS 17)"
  }
}
```

Ce type de structure JSON permet de conserver des informations contextuelles riches (adresse IP, navigateur, identifiants internes, statut de l'inscription, etc.) de manière flexible, tout en séparant clairement les données d'audit des données métiers transactionnelles gérées dans PostgreSQL.

Prévision des migrations et évolution du schéma

L'évolution progressive de la base de données est prise en compte dès la conception, afin d'accompagner les différentes étapes du projet (MVP, version Bêta, version 1). Les migrations seront gérées de manière structurée à l'aide d'outils de versionnement du schéma, permettant de modifier ou d'étendre les tables sans perte de données.
Les principales étapes prévues sont les suivantes :

— ajout de nouvelles colonnes sans interruption de service (ex : données supplémentaires utilisateur) ;

— évolution de certaines tables pour introduire de nouvelles fonctionnalités (ex : paiements et rôles avancés) ;
— création de tables complémentaires lorsque nécessaire (ex : statistiques, logs, modules optionnels) ;
— conservation des données existantes via des scripts de migration contrôlés ;
— traçabilité et possibilité de retour arrière grâce au versionnement du schéma.

Cette approche garantit que la base de données pourra évoluer de manière fiable et contrôlée tout au long du cycle de développement, sans impact sur les utilisateurs ou sur l'intégrité des données.

— PostgreSQL est utilisé pour toutes les données métiers structurées du système (utilisateurs, clubs, adhésions, événements, inscriptions et paiements). Son modèle relationnel garantit l'intégrité, les contraintes, les transactions ACID et la cohérence nécessaire au fonctionnement administratif d'un club de plongée.
— MongoDB est utilisé en complément pour stocker des données non structurées ou volumétriques telles que les logs d'activité, l'audit des actions, les traces techniques ou les retours bruts de l'API HelloAsso. Son modèle documentaire permet une flexibilité d'évolution sans impacter le schéma transactionnel.

L'utilisation combinée des deux systèmes permet donc d'optimiser chaque type de donnée avec la technologie la plus adaptée tout en maintenant des performances et une structure claire.

— PostgreSQL reste la **source de vérité** du système : toutes les données fonctionnelles et métier y sont stockées et validées.
— MongoDB ne stocke que des informations dérivées, techniques ou historiques (logs, audit, réponses externes), et ne contient aucune donnée nécessaire au fonctionnement métier.
— Les documents MongoDB incluent systématiquement des identifiants provenant de PostgreSQL (ex : `userId`, `evenementId`, `inscriptionId`), garantissant la correspondance entre les deux systèmes.
— La suppression ou modification des données métier n'impacte pas l'intégrité du système, car MongoDB n'est jamais utilisé pour prendre une décision fonctionnelle.

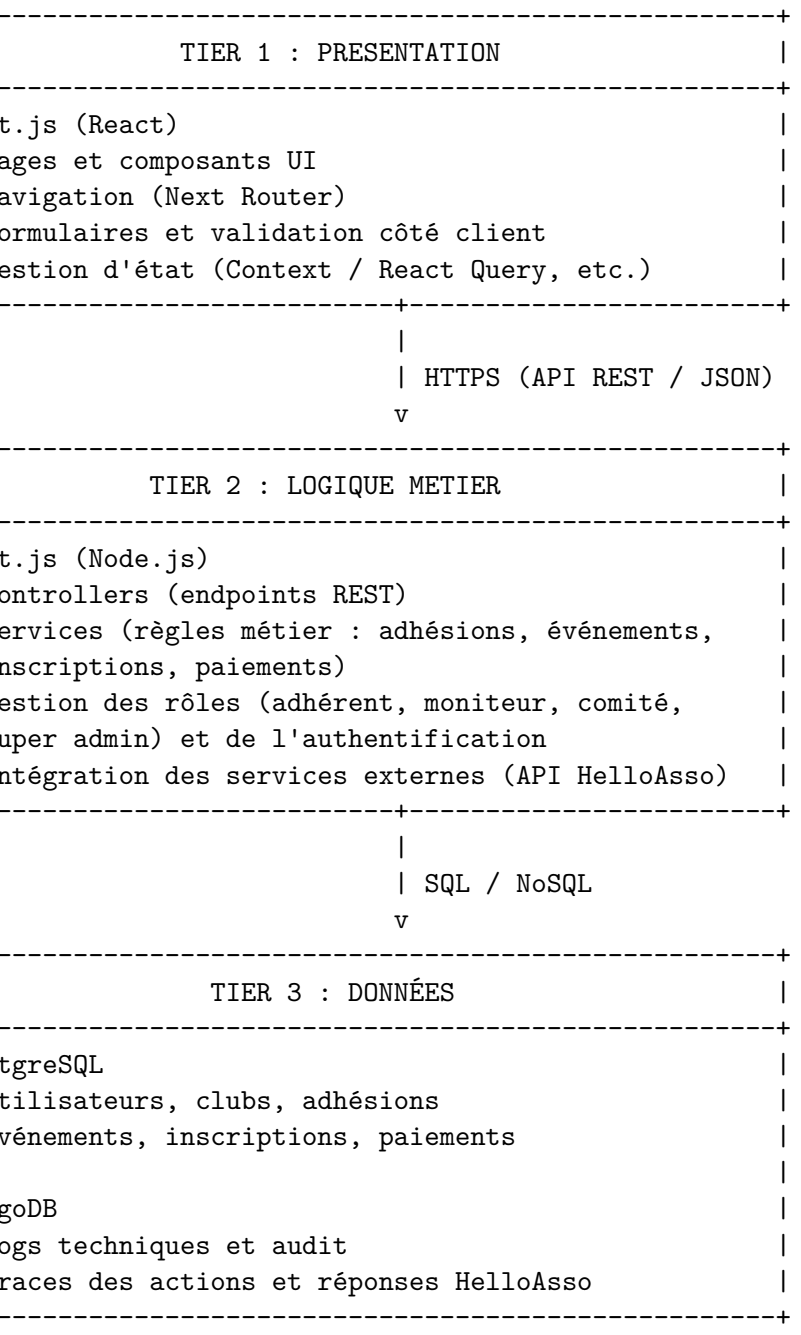
De cette manière, il n'existe pas de risque d'incohérence applicative, et chaque base joue un rôle distinct et maîtrisé dans l'architecture globale.

4.5 Architecture 3 tiers

L'application *Diving O Club* repose sur une architecture 3 tiers, organisée en trois couches clairement séparées : la présentation, la logique métier et les données. Cette approche est particulièrement adaptée à une application web moderne multi-utilisateurs, car elle permet de séparer les responsabilités, de faciliter la maintenance et de faire évoluer chaque couche de manière indépendante. Elle s'intègre naturellement avec la stack technique choisie (Next.js / React pour le front-end, Nest.js / Node.js pour le back-end, PostgreSQL et MongoDB pour les données).

Schéma global de l'architecture

Le schéma ci-dessous illustre l'architecture 3 tiers mise en place pour *Diving O Club* :



Justification de l'architecture choisie

Cette architecture 3 tiers a été retenue pour plusieurs raisons :

— **Séparation claire des responsabilités** : la couche présentation se concentre sur l'interface et l'expérience utilisateur (UX), la couche métier sur les règles propres aux clubs de plongée (adhésions, certificats, rôles, inscriptions aux événements), et la couche données sur la persistance et l'intégrité des informations. Cela limite le couplage entre les parties de l'application.
— **Évolutivité et maintenabilité** : chaque tier peut évoluer indépendamment. Par exemple, l'interface Next.js peut être refondue (nouvelle charte graphique, nouvelles pages) sans réécrire la logique métier, et la base PostgreSQL peut être optimisée (index, migrations) sans impacter le front-end.
— **Alignement avec la stack moderne web** : Next.js (React) s'intègre naturellement avec une API REST Nest.js, ce qui permet d'exposer des endpoints clairs (`/api/clubs`, `/api/events`, etc.) et de gérer l'authentification par jetons. PostgreSQL est utilisé comme base relationnelle robuste pour les données critiques, tandis que MongoDB sert de support flexible pour les logs et l'audit.
— **Facilité de tests et de déploiement** : la séparation des tiers facilite les tests unitaires et d'intégration (tests des services Nest.js indépendamment de l'UI) ainsi que le déploiement sur plusieurs environnements (développement, pré-production, production). L'API peut être testée et validée sans dépendre du front-end.
— **Préparation à la montée en charge** : l'architecture 3 tiers permet, à terme, de faire évoluer l'infrastructure (mise en place de réplicas PostgreSQL, scaling horizontal de Nest.js derrière un reverse proxy, ajout de mécanismes de cache) sans remettre en cause le modèle général.

Cette structuration en trois couches contribue ainsi à rendre *Diving O Club* plus robuste, plus lisible et plus facilement extensible dans la perspective d'une ouverture à plusieurs clubs et d'une évolution fonctionnelle sur le long terme.

4.6 Liens utiles

— UML : https://www.uml-diagrams.org/
— Merise (FR) : https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/intro-merise.html
— OWASP ASVS : https://owasp.org/ASVS/
— PostgreSQL Docs : https://www.postgresql.org/docs/
— MongoDB Modeling : https://bit.ly/mongodb-modeling
— Draw.io (diagrammes) : https://app.diagrams.net/
— Lighthouse (accessibilité) : https://developers.google.com/web/tools/lighthouse
— Lucidchart (UML) : https://www.lucidchart.com/pages/fr/exemples/diagramme-uml
— PlantUML (diagrammes) : https://plantuml.com/
— Mermaid (diagrammes) : https://mermaid-js.github.io/mermaid/

Chapitre 5
Architecture 3 tiers

5.1 Architecture 3 tiers

L'application repose sur une **architecture 3 tiers** classique et éprouvée, séparant strictement l'interface utilisateur, la logique métier et la gestion des données. Ce modèle structure le système en trois couches indépendantes :

— **Couche Présentation** : gère l'affichage, l'interaction et l'expérience utilisateur.
— **Couche Logique Métier** : centralise les règles métier, le contrôle des accès, la validation et les traitements applicatifs.
— **Couche Données** : assure le stockage persistant, l'intégrité, la cohérence et l'accès sécurisé aux informations.

Cette séparation garantit une **meilleure maintenabilité**, une **évolution facilitée**, ainsi qu'une **sécurisation des responsabilités** entre les composants. Chaque couche communique uniquement avec la couche adjacente, ce qui réduit les dépendances, améliore la testabilité (tests unitaires et fonctionnels) et permet de faire évoluer une partie du système sans impacter les autres.

La section suivante détaille précisément le rôle, les responsabilités et les technologies associées à chacune de ces trois couches.

5.1.1 Couche Présentation (Frontend)

Technologies de présentation (Frontend)

— **Langage :** TypeScript (typage léger centré sur le métier).
— **Framework & rendu :** Next.js (React 18 + TypeScript).
— **Gestion d'état global léger :** React Context API.
— **Gestion d'état métier avancé :** Redux Toolkit + React-Redux.
— **UI & composants graphiques :** Material UI (MUI).
— **Routing :** Système de routing natif de Next.js.
— **Appels HTTP :** Axios (client HTTP typé).

La couche présentation de *Diving O Club* s'appuie sur un ensemble de technologies cohérentes et complémentaires, choisies pour garantir à la fois la maintenabilité, les performances et une bonne expérience utilisateur, tout en restant accessibles à mettre en œuvre dans le cadre du projet.

Le **framework Next.js**, basé sur **React 18** et **TypeScript**, sert de socle au frontend. Next.js fournit un système de routing structuré, la possibilité de rendre certaines pages côté serveur et d'optimiser le chargement initial, ce qui est particulièrement adapté à une application consultée fréquemment sur mobile. L'utilisation de **TypeScript raisonnable** permet de typer principalement les entités métier (utilisateur, club, événement, membre, paiement), les propriétés des composants et les réponses d'API. Cela réduit les erreurs au runtime, améliore l'autocomplétion dans l'IDE et assure une cohérence forte avec la couche backend également développée en TypeScript.

La gestion de l'état côté interface est divisée en deux niveaux. Pour l'état global léger et stable (utilisateur connecté, rôle, club courant dans le cadre multi-tenant), l'application utilise le **Context API**, fonctionnalité native de React. Ce mécanisme évite la prolifération de props

(*prop-drilling*) et permet de partager facilement des informations transverses entre plusieurs pages sans ajouter de dépendance supplémentaire.

Pour l'état métier plus complexe, l'application s'appuie sur **Redux Toolkit**, associé à **React-Redux**. Redux Toolkit structure l'état applicatif sous forme de "slices" par domaine (événements, membres, inscriptions, certificats, paiements, équipements) et fournit un ensemble d'outils pour gérer les actions, les reducers et l'asynchronisme de manière standardisée. Ce choix facilite la synchronisation des données entre plusieurs pages, le cache local des données issues de l'API Nest.js et le débogage grâce aux outils de l'écosystème Redux, tout en conservant un code lisible.

L'interface utilisateur repose sur **Material UI (MUI)**, une librairie de composants React conforme aux bonnes pratiques d'accessibilité. MUI fournit des composants prêts à l'emploi (boutons, cartes, formulaires, barres de navigation, tableaux, dialogues) qui sont personnalisés via un thème dédié à *Diving O Club*. Cela permet de gagner du temps de développement, d'assurer une cohérence visuelle entre les différentes pages et de limiter la quantité de CSS spécifique à maintenir.

Le **routing** est géré nativement par Next.js : chaque fichier de page correspond à une route (authentification, tableau de bord, gestion des clubs, événements, inscriptions, administration). La mise en place de layouts partagés (header, navigation, footer) permet de mutualiser la structure visuelle et de réduire la duplication de code. Enfin, les communications avec le backend Nest.js sont centralisées via **Axios**, utilisé comme client HTTP typé. Une instance Axios est configurée dans un dossier `services` (URL de base, en-têtes, token d'authentification), puis chaque domaine fonctionnel expose des fonctions dédiées (authentification, événements, membres, paiements). Cette approche évite de disperser la logique réseau dans les composants et améliore la lisibilité comme la testabilité du frontend.

Structure du frontend

```
src/
+-- app/ ou pages/              # Routing Next.js : pages principales
|   +-- page.tsx / index.tsx    # Page d'accueil / landing
|   +-- login/
|   |   +-- page.tsx           # Page de connexion
|   +-- dashboard/
|       +-- page.tsx           # Tableau de bord
|       +-- events.tsx         # Liste des événements du club
|       +-- members.tsx        # Liste des membres
|       +-- payments.tsx       # Suivi des paiements
|       +-- settings.tsx       # Paramètres du club (admin)
|
+-- components/                # Composants réutilisables
|   +-- layout/
|   |   +-- AppLayout.tsx      # Layout global (header/nav/footer)
|   |   +-- Header.tsx
|   |   +-- Sidebar.tsx
|   |   +-- Footer.tsx
|   |
|   +-- common/                # Composants génériques (MUI wrappers)
|   |   +-- Button.tsx
|   |   +-- TextField.tsx
|   |   +-- Select.tsx
|   |   +-- Modal.tsx
|   |   +-- LoadingSpinner.tsx
```

```
|   |   +-- Alert.tsx
|   |
|   +-- events/                # UI liée aux événements
|   |   +-- EventList.tsx
|   |   +-- EventCard.tsx
|   |   +-- EventForm.tsx
|   |
|   +-- members/               # UI liée aux membres / adhérents
|   |   +-- MemberList.tsx
|   |   +-- MemberCard.tsx
|   |
|   +-- payments/              # UI pour les paiements / HelloAsso
|       +-- PaymentTable.tsx
|       +-- PaymentStatusBadge.tsx
|
+-- context/                   # Context API (état global léger)
|   +-- AuthContext.tsx        # Utilisateur connecté + rôle
|   +-- ClubContext.tsx        # Club courant (multi-tenant)
|
+-- store/                     # Redux (état métier avancé)
|   +-- index.ts               # Configuration du store Redux
|   +-- slices/
|       +-- authSlice.ts       # JWT, profil, permissions front
|       +-- clubSlice.ts       # Infos club courant
|       +-- eventsSlice.ts     # Événements / sorties
|       +-- membersSlice.ts    # Adhérents
|       +-- paymentsSlice.ts   # Paiements / statuts
|
+-- hooks/                     # Hooks personnalisés
|   +-- useAuth.ts             # Accès pratique au contexte Auth
|   +-- useClub.ts             # Accès au club courant
|   +-- useEvents.ts           # Accès aux données d'événements
|
+-- services/                  # Appels API typés vers Nest.js
|   +-- apiClient.ts           # Instance Axios configurée (TypeScript)
|   +-- authService.ts         # Login, refresh token, profil
|   +-- clubService.ts         # Infos club
|   +-- eventService.ts        # CRUD événements
|   +-- memberService.ts       # CRUD membres
|   +-- paymentService.ts      # Intégration HelloAsso / paiements
|
+-- styles/                    # Thème et styles globaux
|   +-- theme.ts               # Thème Material UI (couleurs, typo)
|   +-- globals.css            # Styles globaux Next.js
|
+-- utils/                     # Fonctions utilitaires
    +-- date.ts                # Formatage des dates
    +-- validation.ts          # Validations front simples
    +-- formatters.ts          # Formatage d'affichage
```

5.1.2 Couche Logique Métier (Backend)

La couche logique métier de *Diving O Club* est implémentée avec le framework **Nest.js** en **TypeScript**. Elle constitue le coeur de l'application côté serveur : elle expose une API REST, centralise les règles métier, orchestre les accès aux données et applique les règles de sécurité (authentification, autorisation par rôle, vérification du club dans le cadre multi-tenant). Cette couche est strictement séparée de la présentation (frontend Next.js) et de la persistance (PostgreSQL), conformément à l'architecture 3-tiers présentée au chapitre 4.

L'architecture backend est organisée de manière **modulaire**, avec un module par domaine fonctionnel : `AuthModule` (authentification, JWT, refresh tokens), `ClubsModule` (gestion des clubs et du multi-tenant), `MembersModule` (adhérents et encadrants), `EventsModule` (sorties plongée, séances piscine, voyages), `PaymentsModule` (suivi des paiements et intégration avec HelloAsso), etc. Chaque module regroupe ses propres **controllers**, **services** et **repositories** (DAO), ainsi que les **DTO** (Data Transfer Objects) et les schémas de validation associés.

La gestion des rôles et des droits d'accès repose sur des **guards** Nest.js (par exemple `AuthGuard`, `RolesGuard`) et sur la logique métier des services : les guards vérifient l'authentification et les rôles à partir du JWT avant d'exécuter une route, tandis que les services appliquent les règles métier plus fines (par exemple, un encadrant ne peut créer des événements que pour son propre club). Les **controllers** restent ainsi le plus fins possible et ne gèrent pas directement les rôles.

Controllers

Les **controllers** Nest.js constituent le point d'entrée HTTP de la couche logique métier. Ils reçoivent les requêtes venant du frontend (création d'événement, inscription à une sortie, consultation de la liste des membres, etc.), appliquent les guards d'authentification / autorisation et délèguent immédiatement le traitement aux services. L'objectif est de conserver des contrôleurs fins, limités à l'orientation des requêtes, au mapping des DTO et au choix des codes de retour HTTP.

Chaque contrôleur est décoré avec `@Controller()` pour définir le préfixe de route (par exemple `/events`), et utilise des décorateurs comme `@Get()`, `@Post()`, `@Patch()` ou `@Delete()` pour exposer les différentes opérations du domaine. Les paramètres de route, le corps de la requête et l'utilisateur authentifié sont injectés via des décorateurs (`@Param()`, `@Body()`, `@Req()` ou un décorateur personnalisé pour l'utilisateur courant).

Exemple de contrôleur (simplifié) :

```
import {
  Controller,
  Get,
  Post,
  Body,
  UseGuards
} from '@nestjs/common';
import { EventsService } from './events.service';
import { CreateEventDto } from './dto/create-event.dto';
import { AuthGuard } from '../auth/auth.guard';
import { RolesGuard } from '../auth/roles.guard';
import { Roles } from '../auth/roles.decorator';
import { CurrentUser } from '../auth/current-user.decorator';
import { UserJwtPayload } from '../auth/types';

@Controller('events')
```

```
@UseGuards(AuthGuard, RolesGuard)
export class EventsController {
  constructor(private readonly eventsService: EventsService) {}

  // Création d'un événement (coach ou admin uniquement)
  @Post()
  @Roles('coach', 'admin')
  async create(
    @Body() dto: CreateEventDto,
    @CurrentUser() user: UserJwtPayload,
  ) {
    return this.eventsService.create(dto, user);
  }

  // Liste des événements du club de l'utilisateur connecté
  @Get()
  async findAll(@CurrentUser() user: UserJwtPayload) {
    return this.eventsService.findAllForClub(user.clubId);
  }
}
```

Dans cet exemple, le `EventsController` protège les routes avec des guards (`AuthGuard`, `RolesGuard`) et utilise un décorateur `@Roles` pour indiquer les rôles autorisés sur certaines opérations. Il récupère l'utilisateur courant via `@CurrentUser()` et délègue toute la logique métier au `EventsService`.

Services

Les **services** implémentent la logique métier propre à chaque domaine. Ils orchestrent les différentes opérations : validation métier spécifique, application des règles liées aux clubs (capacité maximale d'une sortie, statut d'un membre, licences et certificats valides), interactions avec plusieurs repositories (événements, membres, paiements), et éventuellement déclenchement de notifications.

Dans *Diving O Club*, chaque module métier expose un service principal (`AuthService`, `ClubsService`, `MembersService`, `EventsService`, `PaymentsService`, etc.). Les services utilisent des DTO typés pour recevoir les données en entrée et retournent des objets métier structurés ou des modèles persistés. Ils encapsulent les règles de gestion afin que le frontend ne fasse jamais de logique sensible côté client.

Exemple de service (simplifié) :

```
import {
  Injectable,
  BadRequestException,
  ForbiddenException
} from '@nestjs/common';
import { EventsRepository } from './events.repository';
import { CreateEventDto } from './dto/create-event.dto';
import { UserJwtPayload } from '../auth/types';

@Injectable()
export class EventsService {
  constructor(private readonly eventsRepository: EventsRepository) {}
```

```
  async create(dto: CreateEventDto, user: UserJwtPayload) {
    // Règle métier 1 : seul un coach ou un admin peut créer un événement
    if (!['coach', 'admin'].includes(user.role)) {
      throw new ForbiddenException('Accès interdit');
    }

    // Règle métier 2 : l'événement appartient toujours au club de l'utilisateur
    dto.clubId = user.clubId;

    // Règle métier 3 : validation métier spécifique
    if (dto.maxParticipants <= 0) {
      throw new BadRequestException('Capacité maximale invalide');
    }

    if (dto.startDate >= dto.endDate) {
      throw new BadRequestException('Dates de début/fin incohérentes');
    }

    // Délégation à la couche d'accès aux données (repository)
    return this.eventsRepository.create(dto);
  }

  async findAllForClub(clubId: number) {
    return this.eventsRepository.findAllByClub(clubId);
  }
}
```

Dans cet exemple, le `EventsService` applique des règles métier liées aux rôles et au contexte : vérification du rôle de l'utilisateur, forçage du club de l'événement, validation de la capacité et de la cohérence des dates. Le service délègue ensuite la persistance au repository.

Repositories (DAO)

Les **repositories** (ou DAO, Data Access Objects) encapsulent l'accès à la base de données. Dans *Diving O Club*, ils s'appuient sur l'ORM **Prisma** pour communiquer avec PostgreSQL. Chaque repository est responsable d'un domaine (événements, membres, clubs, paiements) et expose des méthodes simples : `create`, `findById`, `findAllByClub`, `update`, `delete`, etc.

Ce découpage permet de :

— centraliser les requêtes vers la base de données dans une couche unique ;
— adapter plus facilement la structure de la base sans impacter la logique métier ;
— tester les services indépendamment de la persistance (en simulant les repositories).

Exemple de repository (simplifié) :

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { CreateEventDto } from './dto/create-event.dto';

@Injectable()
export class EventsRepository {
  constructor(private readonly prisma: PrismaService) {}

  async create(dto: CreateEventDto) {
```

```
    return this.prisma.event.create({
      data: {
        title: dto.title,
        description: dto.description,
        startDate: dto.startDate,
        endDate: dto.endDate,
        maxParticipants: dto.maxParticipants,
        clubId: dto.clubId,
        location: dto.location,
      },
    });
  }

  async findAllByClub(clubId: number) {
    return this.prisma.event.findMany({
      where: { clubId },
      orderBy: { startDate: 'asc' },
    });
  }
}
```

Dans cet exemple, `EventsRepository` utilise `PrismaService` pour interagir avec PostgreSQL. Le repository ne contient aucune logique métier : il se limite à mapper les DTO vers le schéma de la base et à exécuter les opérations de persistance. Les règles métier restent dans les services, et les contrôleurs se concentrent sur la gestion des requêtes HTTP. Cette organisation **Controller** → **Service** → **Repository** respecte la séparation des responsabilités attendue dans une architecture 3-tiers et prépare l'application à évoluer sans remettre en cause l'ensemble du backend.

5.1.3 Couche Données (Database)

Dans cette sous-section, la couche de données de *Diving O Club* est décrite en termes d'architecture, de choix techniques et de rôle dans l'architecture 3 tiers. L'objectif est de proposer une base fiable pour les opérations transactionnelles (adhérents, événements, inscriptions, paiements) tout en conservant la possibilité d'exploiter des logs et rapports techniques pour le suivi de l'application.

L'application s'appuie principalement sur une base **PostgreSQL** pour toutes les données métier structurées (clubs, utilisateurs, membres, événements, inscriptions, certificats, paiements). PostgreSQL offre un modèle relationnel robuste, des contraintes d'intégrité (clés étrangères, unicité, contraintes de vérification) et un langage SQL riche, adapté à la gestion des relations complexes entre entités (par exemple un membre rattaché à un club, à plusieurs événements et à plusieurs paiements).

Les accès à PostgreSQL sont réalisés via l'ORM **Prisma**, qui fournit des modèles fortement typés en TypeScript et un système de migrations pour faire évoluer le schéma de données de manière contrôlée. Prisma simplifie l'écriture des requêtes tout en conservant un contrôle explicite sur les relations et les jointures.

En complément, une base **MongoDB** peut être utilisée pour stocker des logs techniques ou des rapports non-structurés, lorsque cela est nécessaire (suivi d'activité, traces, journalisation applicative). Dans ce cas, les données MongoDB ne portent pas de logique métier critique et ne sont pas utilisées dans les parcours principaux de l'utilisateur.

Architecture des données :

— **PostgreSQL** : Données transactionnelles et relations (clubs, membres, événements, inscriptions, paiements).

— **MongoDB** : Logs et rapports techniques non-structurés (optionnel).
— **ORM** : Prisma pour PostgreSQL (modèles typés, migrations).

Exemple de repository PostgreSQL (Prisma) :

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { CreateEventDto } from './dto/create-event.dto';

@Injectable()
export class EventsRepository {
  constructor(private readonly prisma: PrismaService) {}

  async create(dto: CreateEventDto) {
    return await this.prisma.event.create({
      data: {
        title: dto.title,
        description: dto.description,
        startDate: dto.startDate,
        endDate: dto.endDate,
        maxParticipants: dto.maxParticipants,
        clubId: dto.clubId,
        location: dto.location,
      },
      include: {
        club: {
          select: { id: true, name: true },
        },
      },
    });
  }
}
```

5.1.4 Communication entre les tiers

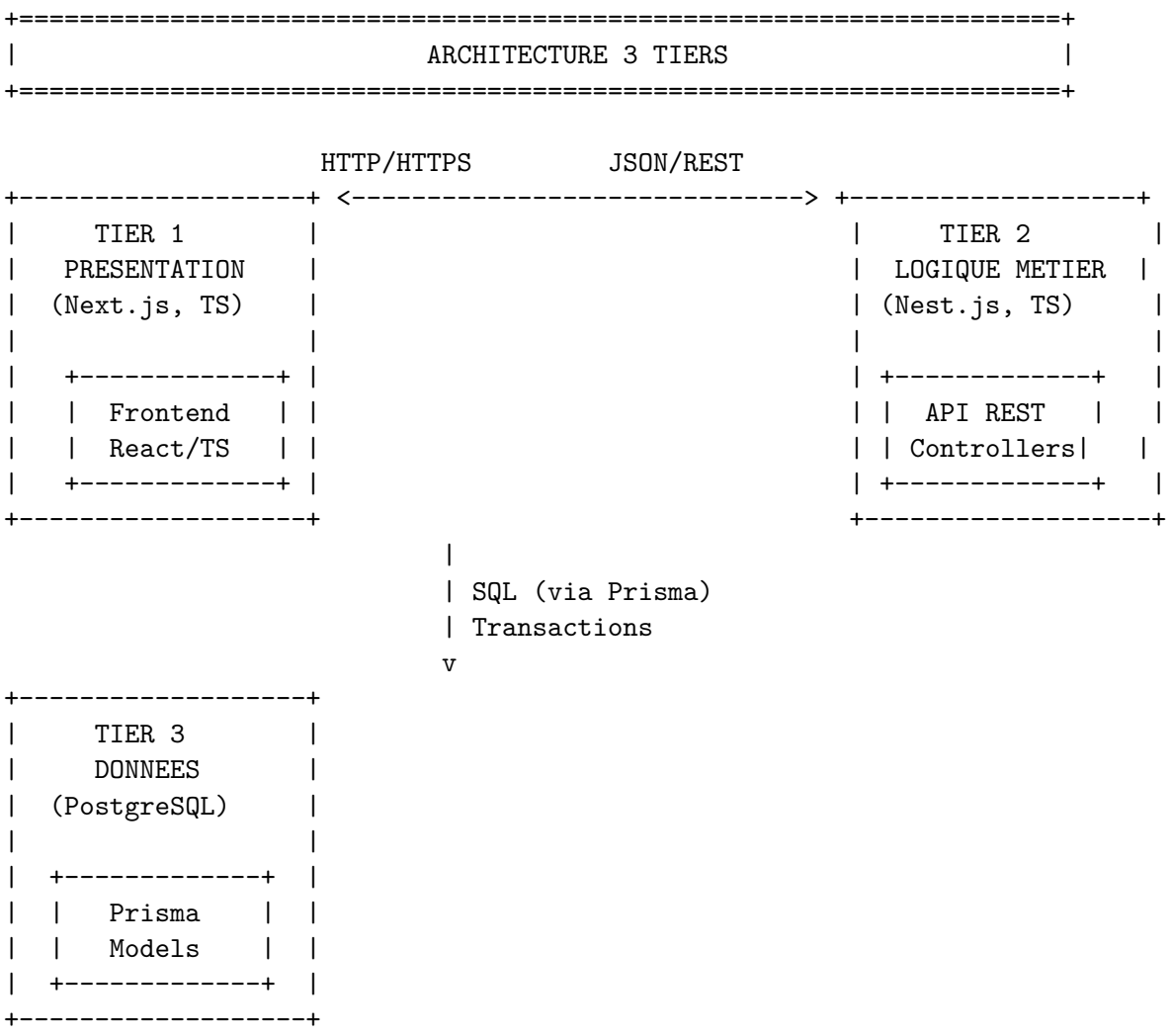
5.1.4 Communication entre les tiers

Dans cette sous-section, la communication entre les trois couches de l'architecture est décrite afin de montrer comment les données circulent dans *Diving O Club*. L'objectif est de garantir une séparation claire des responsabilités entre la présentation, la logique métier et la couche données, tout en assurant des échanges sécurisés et standardisés.

La communication entre le **frontend** (Next.js) et le **backend** (Nest.js) repose sur des appels **HTTP/HTTPS** utilisant un format d'échange standardisé en **JSON**. Le frontend envoie des requêtes vers les endpoints REST exposés par l'API Nest.js (exemples : `/auth/login`, `/events`, `/members`). Chaque requête utilise les tokens d'authentification transmis dans les en-têtes afin de garantir la sécurité (JWT).

Le backend applique les rules d'accès via des **guards**, valide les données via des DTO, exécute la logique métier dans les services, puis délègue l'accès aux données aux repositories. La communication entre le backend et la couche données repose sur les requêtes SQL générées par Prisma, exécutées dans PostgreSQL. Ces interactions sont transactionnelles, fortement typées et assurent l'intégrité des données (relations, contraintes, cohérence).

Flux de communication 3 tiers :



Dans cette architecture, chaque tier reste indépendant : la présentation ne contient aucune logique métier, le backend ne connaît pas l'interface graphique, et la base de données est isolée de l'application via le pattern Repository. Les échanges sont sécurisés, validés et parfaitement structurés pour garantir la cohérence globale du système.

5.1.5 Avantages de l'architecture 3 tiers

5.1.5 Avantages de l'architecture 3 tiers

L'architecture 3 tiers mise en place pour *Diving O Club* présente plusieurs avantages majeurs en termes de clarté, de sécurité, de performance et d'évolutivité. La séparation nette entre la couche présentation, la couche logique métier et la couche données permet d'isoler les responsabilités et de maintenir une organisation cohérente du code.

— **Séparation des responsabilités** : chaque tier remplit un rôle précis. Le frontend (Next.js) gère l'affichage et les interactions, le backend (Nest.js) centralise la logique métier, et la base de données (PostgreSQL) assure la persistance fiable des données. Cette séparation rend l'architecture plus lisible et facilite les évolutions futures.

— **Maintenabilité accrue** : les modifications apportées à un tier n'impactent pas directement les autres. Par exemple, faire évoluer l'interface utilisateur n'exige aucune modification du backend, et ajuster une règle métier n'implique pas de modifier les composants graphiques. Cette isolation réduit fortement les risques de régressions.

— **Scalabilité** : chaque tier peut être optimisé ou dimensionné indépendamment. Le frontend peut être déployé sur un CDN, le backend peut être répliqué derrière un load balancer, et

la base PostgreSQL peut évoluer (indexation, montée en charge, réplications). Cette modularité permet d'adapter l'application si plusieurs clubs utilisent la plateforme en parallèle (multi-tenant).

— **Sécurité renforcée** : le backend impose des contrôles d'accès stricts (JWT, rôles, guards Nest.js), et la base est isolée via des repositories qui empêchent l'accès direct depuis le frontend. Les données critiques (membres, certificats, événements, paiements) ne sont accessibles qu'à travers des règles métier validées.
— **Performance optimisée** : l'API REST peut être optimisée indépendamment du frontend, les requêtes SQL peuvent être indexées ou réécrites sans changer le code métier, et le frontend peut bénéficier du rendu serveur de Next.js pour améliorer l'expérience utilisateur sur mobile.
— **Facilité de tests** : chaque tier peut être testé séparément : tests unitaires sur les services Nest.js, tests d'intégration sur les repositories, tests end-to-end sur l'API, et tests d'interface sur les composants React. Cette séparation améliore la qualité globale du projet.

Cette architecture permet ainsi à *Diving O Club* de rester cohérente, évolutive, sécurisée et adaptée à un contexte multi-tenant où plusieurs clubs peuvent utiliser simultanément la plateforme.

5.2 Développement Frontend

5.2 Développement Frontend

Dans cette section, le développement frontend de *Diving O Club* est présenté sous l'angle des choix techniques, de l'architecture des composants, ainsi que des bonnes pratiques d'accessibilité et de performances. L'objectif est de proposer une interface utilisateur moderne, responsive, fiable et adaptée à une utilisation mobile-first, comme exigé dans le cahier des charges.

Technologies choisies

Le frontend repose sur le framework **Next.js** (React 18) avec **TypeScript**. Cette technologie a été choisie pour plusieurs raisons :

— **Performances** : le rendu serveur (SSR) et le pré-rendu (SSG) améliorent les temps de chargement, notamment sur mobile.
— **Sécurité et robustesse** : TypeScript réduit les erreurs grâce au typage statique et à l'autocomplétion.
— **État global** : Redux Toolkit permet de centraliser l'état utilisateur (authentification, club courant, rôle, tokens).
— **UI professionnelle** : Material UI (MUI) assure une cohérence visuelle et des composants accessibles.
— **Communication API** : Axios est utilisé pour consommer l'API REST du backend.
— **Routing avancé** : le routeur intégré de Next.js facilite la navigation et le découpage des pages.

Architecture des composants

L'architecture du frontend suit une organisation modulaire inspirée des bonnes pratiques React, avec une séparation claire entre composants réutilisables, hooks, services, stores et pages. Cette organisation améliore la maintenabilité et facilite la réutilisation du code.

```
src/
+-- components/             # Composants réutilisables
|   +-- common/             # Composants génériques
```

```
|   |   +-- Button.tsx
|   |   +-- Modal.tsx
|   |   +-- LoadingSpinner.tsx
|   |
|   +-- auth/               # Pages et vues liées à l'authentification
|   |   +-- LoginForm.tsx
|   |   +-- RegisterForm.tsx
|   |
|   +-- events/             # Fonctionnalité événements
|   |   +-- EventCard.tsx
|   |   +-- EventList.tsx
|   |   +-- EventForm.tsx
|   |
|   +-- members/            # Fonctionnalité membres
|       +-- MemberCard.tsx
|       +-- MemberList.tsx
|
+-- pages/                  # Pages Next.js
+-- store/                  # Redux Toolkit (slices + store global)
+-- hooks/                  # Hooks personnalisés
+-- services/               # Appels API (Axios)
+-- utils/                  # Fonctions utilitaires
```

Cette organisation par domaine fonctionnel (Events, Members, Auth, Clubs) permet un découplage clair, facilite les tests et réduit la duplication de code. Les composants sont pensés pour être réutilisables (boutons, formulaires, cartes, listes, modales), tandis que les vues composent ces briques pour construire les pages finales.

Accessibilité et UX

L'application respecte les bonnes pratiques RGAA/WCAG afin d'assurer une expérience inclusive : contrastes suffisants, gestion des focus, attributs ARIA, composants accessibles via Material UI, navigation clavier et alternatives textuelles.

Des audits automatisés via **Lighthouse** ont été réalisés pour mesurer les performances, l'accessibilité et les bonnes pratiques. Les scores obtenus montrent une interface optimisée sur mobile et desktop.

Exemple de composant accessible :

```
const EventCard = ({ event, onOpen }) => {
  return (
    <div
      className="event-card"
      role="article"
      aria-labelledby={`event-${event.id}-title`}
    >
      <h3 id={`event-${event.id}-title`}>{event.title}</h3>
      <p>{event.description}</p>

      <button
        onClick={() => onOpen(event.id)}
        aria-label={`Voir les détails de ${event.title}`}
      >
        Détails
```

```
      </button>
    </div>
  );
};
```

Exemple de rapport Lighthouse :

```
{
  "categories": {
    "performance": { "score": 0.90 },
    "accessibility": { "score": 0.96 },
    "best-practices": { "score": 0.89 },
    "seo": { "score": 0.92 }
  }
}
```

Tests, qualité et sécurité

— Les composants sont testés avec **Jest** et **React Testing Library**.
— La protection XSS est assurée grâce au modèle de rendu de React et aux entrées validées par le backend.
— L'état global est typé avec TypeScript pour limiter les incohérences.
— Les performances sont contrôlées via Lighthouse et l'optimisation des assets (images, fonts, lazy loading).

L'approche frontend proposée garantit une interface fluide, accessible, mobile-first et capable d'évoluer pour accueillir plusieurs clubs dans un contexte multi-tenant.

5.3 Développement Backend

5.3 Développement Backend

Le backend de *Diving O Club* est développé avec le framework **Nest.js** en TypeScript. Il expose une API REST structurée selon le pattern **Controller / Service / Repository**, garantissant une séparation claire des responsabilités. Les contrôleurs gèrent les requêtes HTTP, les services encapsulent la logique métier, et les repositories (DAO) assurent l'accès aux données via Prisma et PostgreSQL.

Nest.js fournit une architecture modulaire qui facilite l'organisation par domaines fonctionnels : `AuthModule`, `EventsModule`, `MembersModule`, `ClubsModule`, `PaymentsModule`. Chaque module contient ses propres contrôleurs, services, DTO et repositories, ce qui renforce la cohérence globale de l'application.

Validation des données

Les données entrantes sont validées via les **DTO** (Data Transfer Objects) de Nest.js, associés à des decorators comme `@IsString`, `@IsEmail`, `@IsInt`, etc., fournis par la bibliothèque **class-validator**. Cette approche garantit la conformité des données dès la réception de la requête, avant l'exécution de la logique métier.

Une validation incorrecte retourne automatiquement une erreur HTTP 400 grâce au **ValidationPipe** global de Nest.js, assurant une gestion homogène des réponses d'erreur.

Gestion des erreurs

Nest.js utilise un système d'exceptions centralisé basé sur les classes telles que `BadRequestException`, `UnauthorizedException`, `ForbiddenException` ou `NotFoundException`. Une erreur levée dans

un service est interceptée et transformée en réponse JSON standardisée. Cette approche simplifie le debugging et garantit des réponses cohérentes entre les différents endpoints.

Authentification et sécurité

L'authentification repose sur des tokens **JWT** (access token et refresh token). Les endpoints sensibles sont protégés via des **guards** Nest.js :

— `AuthGuard` : vérifie la présence et la validité du token JWT.

— `RolesGuard` : vérifie le rôle et les permissions de l'utilisateur.

Les tokens contiennent uniquement les informations minimales nécessaires : `id`, `role`, `clubId`, assurant la compatibilité avec l'approche multi-tenant.

Une fois authentifié, l'utilisateur peut accéder aux opérations autorisées pour son rôle : membres, encadrants, administrateurs du club.

Documentation API

L'API est documentée avec **Swagger** (OpenAPI), intégré nativement à Nest.js. La documentation comprend :

— la liste des endpoints disponibles,

— les paramètres attendus,

— les DTO utilisés,

— les schémas de réponses,

— les erreurs possibles.

Cette documentation facilite l'intégration frontend, la communication avec les futurs développeurs et les tests end-to-end.

Structure du backend

```
src/
+-- auth/
|   +-- auth.controller.ts
|   +-- auth.service.ts
|   +-- auth.repository.ts
|   +-- dto/
|
+-- events/
|   +-- events.controller.ts
|   +-- events.service.ts
|   +-- events.repository.ts
|   +-- dto/
|
+-- members/
|   +-- members.controller.ts
|   +-- members.service.ts
|   +-- members.repository.ts
|   +-- dto/
|
+-- prisma/
|   +-- prisma.service.ts
|
+-- guards/
```

```
|   +-- auth.guard.ts
|   +-- roles.guard.ts
|
+-- main.ts
```

Cette structure met en évidence la séparation des responsabilités : les contrôleurs sont fins, les services regroupent les règles métier, et les repositories assurent l'interaction avec la couche données.

Exemple de contrôleur Nest.js avec validation et sécurité

```
@Post()
@UseGuards(AuthGuard, RolesGuard)
@Roles('coach', 'admin')
async createEvent(
  @Body() dto: CreateEventDto,
  @CurrentUser() user
) {
  return await this.eventsService.create(dto, user);
}
```

Cet exemple illustre le fonctionnement complet : validation via DTO, protection via guards, et délégation de la logique métier au service.

À faire / À vérifier

— Séparer clairement les responsabilités (Controller / Service / Repository).
— Valider systématiquement les données via DTO + ValidationPipe.
— Utiliser une gestion centralisée des erreurs.
— Documenter l'API avec Swagger (OpenAPI).
— Logger les actions importantes (créations, suppressions, erreurs).

Ce backend modulaire, typé et sécurisé permet à *Diving O Club* d'évoluer vers un environnement multi-tenant tout en garantissant lisibilité, maintenabilité et robustesse.

5.4 Gestion des données

5.4 Gestion des données

La couche données de *Diving O Club* repose principalement sur une base **PostgreSQL**, utilisée pour gérer l'ensemble des données métier critiques : utilisateurs, clubs, membres, événements, inscriptions, certificats, paiements, rôles, etc. PostgreSQL offre un modèle relationnel robuste, des transactions fiables et des contraintes d'intégrité strictes (clés étrangères, unicité, règles de cohérence) indispensables pour une application multi-tenant.

L'accès aux données PostgreSQL est réalisé via l'ORM **Prisma**. Celui-ci génère des modèles fortement typés en TypeScript, simplifie la rédaction des requêtes et applique un système de migrations cohérent avec l'évolution du schéma. Le pattern Repository utilisé dans l'architecture backend isole complètement la logique métier de la persistance afin de faciliter les tests et la maintenabilité.

En complément, une base **MongoDB** peut être utilisée pour stocker des logs techniques, rapports ou données non-structurées (par exemple l'activité du club ou les événements système). MongoDB est adapté aux données volumineuses ou faiblement structurées, et ses pipelines d'agrégation permettent d'effectuer des analyses avancées sans impacter la base transactionnelle.

La séparation entre données relationnelles (PostgreSQL) et données analytiques ou techniques (MongoDB) optimise les performances et garantit la stabilité des opérations critiques.

Les transactions Prisma assurent la cohérence lors d'opérations complexes, comme l'inscription d'un membre à un événement ou la validation d'un paiement.

Exemple de repository PostgreSQL (Prisma)

```
@Injectable()
export class EventsRepository {
  constructor(private readonly prisma: PrismaService) {}

  async create(dto: CreateEventDto) {
    return await this.prisma.event.create({
      data: {
        title: dto.title,
        description: dto.description,
        startDate: dto.startDate,
        endDate: dto.endDate,
        maxParticipants: dto.maxParticipants,
        clubId: dto.clubId,
        location: dto.location
      },
      include: {
        club: { select: { id: true, name: true } },
        subscriptions: true
      }
    });
  }

  async findByClub(clubId: number) {
    return await this.prisma.event.findMany({
      where: { clubId },
      include: { subscriptions: true }
    });
  }
}
```

Cet exemple montre comment Prisma permet de gérer les relations SQL et de structurer les retours sans exposer la base directement aux services.

Exemple de pipeline MongoDB (logs techniques)

```
// Pipeline d'agrégation pour les statistiques d'activité
const getEventStats = async (eventId, dateRange) => {
  return await activityLogs.aggregate([
    {
      $match: {
        'metadata.eventId': eventId,
        timestamp: { $gte: dateRange.start, $lte: dateRange.end }
      }
    },
    {
      $group: {
        _id: '$action',
        count: { $sum: 1 },
```

```
        uniqueUsers: { $addToSet: '$userId' }
      }
    },
    {
      $project: {
        action: '$_id',
        count: 1,
        uniqueUsersCount: { $size: '$uniqueUsers' }
      }
    }
  ]);
};
```

L'usage de MongoDB est optionnel dans *Diving O Club*, mais permet d'externaliser les analyses volumineuses sans impacter les performances PostgreSQL.

À faire / À vérifier

— Utiliser un ORM (Prisma) pour simplifier et typer les requêtes SQL.
— Optimiser les requêtes via des index appropriés (recherches par club, par utilisateur).
— Implémenter des transactions pour les opérations critiques.
— Séparer données transactionnelles (PostgreSQL) et analytiques (MongoDB).
— Tester les requêtes sur des jeux de données réalistes.

Cette gestion des données garantit la cohérence, la performance et l'évolutivité de *Diving O Club*, notamment dans un contexte multi-tenant où plusieurs clubs utilisent la plateforme simultanément.

5.5 Liens utiles

— OpenAPI/Swagger : https://swagger.io/specification/
— WCAG : https://www.w3.org/WAI/standards-guidelines/wcag/
— Lighthouse : https://developers.google.com/web/tools/lighthouse
— PostgreSQL Tutorial : https://www.postgresql.org/docs/current/tutorial.html
— MongoDB Aggregation : https://www.mongodb.com/docs/manual/aggregation/
— Prisma Documentation : https://www.prisma.io/docs/

Chapitre 6
Sécurité applicative et RGPD

6.1 Protection contre les vulnérabilités OWASP (Top 10 : 2025)

La sécurité de l'application *Diving O Club* s'appuie sur les recommandations du **OWASP Top 10 : 2025**. Cette section présente les vulnérabilités ciblées, les mesures de protection intégrées et la gestion sécurisée des flux entre les tiers. L'objectif est de réduire la surface d'attaque, d'assurer l'intégrité des données et de garantir un fonctionnement résilient même face à des comportements malveillants.

6.1.1 Vulnérabilités OWASP ciblées

Les protections implémentées adressent spécifiquement les 10 catégories du OWASP 2025 :

— **A01 :2025 — Broken Access Control** : contrôles d'accès stricts via les *Guards* Nest.js, isolation multi-tenant stricte sur le `clubId` et restrictions basées sur les rôles.

— **A02 :2025 — Security Misconfiguration** : configuration sécurisée de Nest.js, désactivation des messages d'erreur internes, utilisation de `Helmet`, et configuration stricte des CORS.

— **A03 :2025 — Software Supply Chain Failures** : gestion stricte des dépendances npm, analyse des vulnérabilités via `npm audit`, mises à jour régulières, et vérification de l'intégrité des packages.

— **A04 :2025 — Cryptographic Failures** : chiffrement des mots de passe avec Argon2, chiffrement TLS pour toutes les communications et gestion sécurisée des tokens JWT.

— **A05 :2025 — Injection** : utilisation systématique de Prisma pour prévenir les injections SQL, absence de concaténation de chaînes dans les requêtes, et sanitisation des entrées utilisateur.

— **A06 :2025 — Insecure Design** : architecture en couches (3 tiers), séparation stricte des responsabilités, principe du moindre privilège et absence de logique métier côté frontend.

— **A07 :2025 — Authentication Failures** : implémentation de tokens courts, rotation des refresh tokens, contrôle strict des accès et gestion des erreurs d'authentification.

— **A08 :2025 — Software or Data Integrity Failures** : migrations Prisma contrôlées, absence de mise à jour dynamique de scripts côté client, et vérifications d'intégrité lors des opérations sensibles.

— **A09 :2025 — Logging & Alerting Failures** : traçabilité complète des actions critiques, logs structurés, audit des événements majeurs (connexion, création d'événement, modification de rôle) et surveillance centralisée.

— **A10 :2025 — Mishandling of Exceptional Conditions** : gestion unifiée des exceptions Nest.js, masquage des erreurs internes et réponses standardisées entre les différents tiers.

6.1.2 Mesures de protection mises en place

Pour se prémunir contre ces vulnérabilités, plusieurs mesures défensives ont été intégrées :

— **Helmet** pour configurer les headers de sécurité (HSTS, CSP, XSS-Protection).

— **CORS strict** afin de limiter les appels cross-origin non autorisés.

— **Rate limiting** pour limiter les attaques par brute force ou DoS.

— **Validation systématique des entrées** via DTO + `ValidationPipe`.

— **Sanitisation des données** pour prévenir les injections et attaques XSS.

— **Contrôles d'accès exhaustifs** avec *AuthGuard* et *RolesGuard*.
— **Restriction de surface d'exposition** : aucune route sensible sans authentification.
— **Sécurisation du transport** : API disponible uniquement via HTTPS.

6.1.3 Validation des entrées et gestion des erreurs

La validation et la gestion des erreurs jouent un rôle essentiel dans la protection contre les vulnérabilités OWASP :

— **Validation** : les données entrantes sont vérifiées selon les DTO Nest.js, empêchant données incorrectes ou malveillantes d'atteindre la logique métier.
— **Gestion des exceptions** : Nest.js centralise la gestion des erreurs avec des classes telles que `BadRequestException`, `UnauthorizedException`, `ForbiddenException`, ou `InternalServerErrorExc`
— **Masquage des informations internes** : aucune stack trace ni détail sensible n'est renvoyé au client.
— **Standardisation des réponses** : structure cohérente des erreurs pour faciliter l'intégration frontend.

6.1.4 Sécurisation des flux entre tiers

Les communications entre frontend, backend et base de données sont strictement protégées :

— **Frontend** $\rightarrow$ **Backend** : communication en HTTPS, utilisation du header `Authorization: Bearer <token>` et absence de données sensibles dans les URLs.
— **Backend** $\rightarrow$ **Base de données** : Prisma garantit des requêtes paramétrées et une isolation transactionnelle forte.
— **Éviction des données inutiles** : le backend ne retourne que les données strictement nécessaires (principe de minimisation).

6.1.5 Scalabilité et protection contre les attaques DoS

L'architecture backend étant entièrement **stateless**, l'application peut facilement monter en charge :

— **Stateless JWT** : ne nécessite aucune session serveur, ce qui permet de déployer plusieurs instances de Nest.js derrière un load balancer.
— **Rate limiting** : empêche la surcharge du serveur par un trop grand nombre de requêtes provenant d'une même source.
— **Séparation des tiers** : chaque couche peut être scalée indépendamment.

Ces protections combinées assurent que *Diving O Club* reste conforme aux bonnes pratiques de sécurité en vigueur pour l'année 2025 et qu'elle résiste efficacement aux vulnérabilités identifiées par l'OWASP.

6.2 Authentification et autorisation

L'application *Diving O Club* utilise un système d'authentification moderne basé sur des tokens JWT, conforme aux bonnes pratiques de l'OWASP 2025 (A07 Authentication Failures). Cette section décrit les mécanismes de connexion, la gestion des mots de passe, les rôles et permissions, ainsi que les protections associées.

6.2.1 Authentification JWT

L'authentification repose sur un jeton d'accès (*access token*) à courte durée de vie, complété par un *refresh token* permettant d'obtenir un nouveau jeton sans redemander les identifiants. Les tokens sont générés et vérifiés côté backend via les services Nest.js.

— **Access token court** : durée limitée afin de réduire l'impact d'un vol de jeton.
— **Refresh token** : stocké de manière sécurisée et régénéré à chaque renouvellement.
— **Transport sécurisé** : les tokens sont envoyés dans l'en-tête `Authorization: Bearer <token>`.
— **Déconnexion sécurisée** : invalidation du refresh token lors de la déconnexion.

6.2.2 Sécurisation des mots de passe

Pour protéger les comptes utilisateur, l'application utilise **Argon2**, un algorithme de hachage moderne spécialement conçu pour résister aux attaques par force brute ou GPU. Aucun mot de passe n'est jamais stocké en clair en base de données.

— **Argon2id** pour le hachage.
— **Salage automatique** intégré.
— **Politique de réauthentification** pour les actions sensibles.

6.2.3 Gestion des rôles et permissions

L'application implémente un contrôle d'accès basé sur les rôles (RBAC). Chaque utilisateur appartient à un club et possède un rôle déterminant ses autorisations :

— **admin** : gestion du club, des membres, des événements.
— **coach** : gestion des événements et des inscriptions.
— **member** : consultation et inscription aux activités.

L'autorisation est appliquée via les `Guards` Nest.js :

— **AuthGuard** : vérifie la présence d'un JWT valide.
— **RolesGuard** : vérifie que le rôle de l'utilisateur autorise l'opération demandée.
— **Contrôle multi-tenant** : l'utilisateur ne peut agir que dans son propre club.

6.2.4 Gestion des erreurs d'authentification

Les erreurs d'authentification et d'autorisation sont traitées de manière centralisée afin d'éviter les fuites d'informations sensibles et d'assurer une cohérence entre les tiers.

— `UnauthorizedException` pour les tokens invalides ou expirés.
— `ForbiddenException` pour les actions non autorisées.
— **Messages d'erreur neutres** afin de ne pas révéler si un compte existe ou non.
— **Journalisation des tentatives** : tentative de connexion, changement de mot de passe.

Ces mécanismes garantissent une authentification robuste, granulaire et conforme aux normes modernes de sécurité pour l'année 2025.

6.3 Conformité RGPD

L'application *Diving O Club* traite des données personnelles appartenant aux membres des clubs (identité, informations de contact, certificats de plongée, inscriptions aux événements). Conformément au Règlement Général sur la Protection des Données (RGPD), des mesures techniques et organisationnelles ont été mises en place pour garantir la licéité, la sécurité et la transparence du traitement.

6.3.1 Registre des traitements

Un registre décrit l'ensemble des traitements réalisés par l'application : collecte, finalité, base légale, durée de conservation et catégories de données traitées. Les principales finalités sont :

— gestion des comptes utilisateur,
— gestion des membres d'un club,
— gestion des événements et inscriptions,
— suivi des qualifications et certificats.

La base légale retenue est l'**intérêt légitime du club** pour la gestion de ses activités.

6.3.2 Droits RGPD des utilisateurs

L'application implémente les principaux droits prévus par le RGPD :

— **Droit d'accès** : chaque utilisateur peut consulter ses informations personnelles.
— **Droit de rectification** : possibilité de mettre à jour ses données.
— **Droit d'effacement** : suppression du compte sur demande.
— **Droit à la portabilité** : export des données disponibles dans un format structuré.
— **Droit d'opposition** : possibilité de demander la suppression ou l'arrêt du traitement.

Ces actions sont réalisées via des routes sécurisées nécessitant une authentification valide.

6.3.3 Sécurité des données personnelles

Les données personnelles sont protégées par des mécanismes techniques conformes au RGPD :

— **Chiffrement en transit** via HTTPS.
— **Hachage Argon2** des mots de passe stockés en base.
— **Minimisation des données** : seules les informations nécessaires sont conservées.
— **Séparation des rôles** pour limiter l'accès aux données sensibles.
— **Logs d'accès et d'actions critiques** pour assurer la traçabilité (création d'événement, modification de rôle, mise à jour de profil).

6.3.4 Conservation et suppression des données

Les données personnelles sont conservées uniquement pour la durée nécessaire à la gestion du club. Lorsqu'un utilisateur quitte un club ou demande la suppression de ses données :

— son compte est supprimé,
— ses inscriptions et données non essentielles sont anonymisées,
— les logs de sécurité sont conservés pour la durée légale en les rendant non identifiables.

6.3.5 Notification des violations

En cas de violation de données personnelles :

— un rapport d'incident est généré,
— les administrateurs du club sont notifiés,
— une notification peut être transmise à la CNIL si nécessaire.

L'objectif est de garantir transparence et réactivité conformément au RGPD.

Cette approche assure que *Diving O Club* respecte les obligations légales relatives à la protection des données, en combinant sécurité, transparence et contrôle utilisateur.

Chapitre 7

Tests et qualité logicielle

Ce chapitre présente la stratégie globale de tests mise en place pour assurer la fiabilité, la performance et la qualité du projet *Diving O Club*. La couverture fonctionnelle repose sur une pyramide de tests équilibrée : une base solide de tests unitaires pour valider la logique métier, des tests d'intégration pour vérifier les interactions entre les différents modules de l'application, et des tests end-to-end pour garantir le bon fonctionnement des parcours utilisateurs clés. Cette organisation permet d'assurer une couverture élevée tout en optimisant le temps d'exécution.

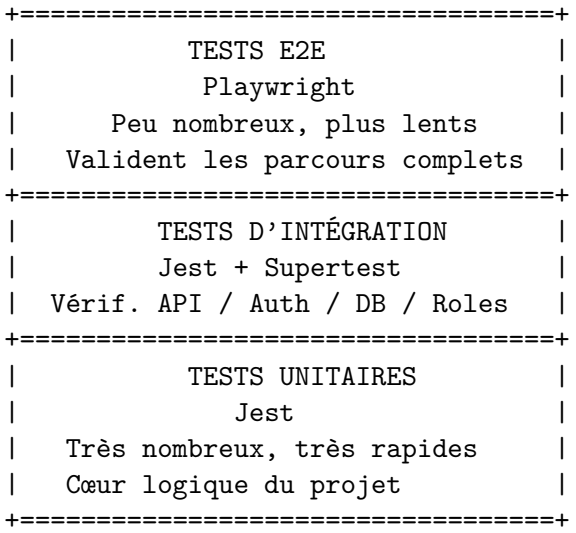
Les tests de performance mesurent la latence (P95) et la capacité de l'application à supporter la charge, tandis que les tests de sécurité automatisés identifient les vulnérabilités courantes. Enfin, la qualité du code est suivie en continu grâce à SonarQube, afin de maintenir un niveau de qualité homogène et maîtrisé tout au long du cycle de développement.

7.1 Stratégie de tests

La stratégie de tests de *Diving O Club* s'appuie sur le modèle de la pyramide de tests, un standard reconnu permettant d'assurer une qualité maximale tout en maîtrisant les coûts d'exécution. Ce modèle hiérarchise les tests selon leur rapidité, leur fiabilité et leur portée :

— **Tests unitaires** : nombreux, rapides, automatisables, ils forment la base du système.
— **Tests d'intégration** : moins nombreux, ils valident la cohérence entre plusieurs modules.
— **Tests End-to-End (E2E)** : rares mais essentiels, couvrant des scénarios complets utilisateurs.

Pyramide de tests :



Cette approche garantit un niveau élevé de qualité tout en permettant d'identifier les erreurs le plus tôt possible dans le cycle de développement.

7.1.1 Tests unitaires

Les tests unitaires constituent la base de la pyramide. Ils vérifient de manière isolée le comportement des services, helpers, pipes ou règles métier du backend NestJS. Ils permettent de détecter rapidement des régressions dans la logique applicative.

Pourquoi réaliser des tests unitaires ?

— **Rapidité** : ils s'exécutent en quelques millisecondes.
— **Fiabilité** : ils valident les règles métier isolées de toute dépendance externe.
— **Détection précoce** : la majorité des erreurs viennent d'une mauvaise logique métier.
— **Sécurité évolutive** : ils évitent que des modifications futures cassent des fonctionnalités stables.

Valeur ajoutée des tests unitaires

— Stabilisation des règles propres au projet multi-tenant.
— Vérification des comportements critiques (calculs, validations, permissions).
— Réduction des temps de debugging.

Exemple 1 : filtrage des clubs visibles dans la recherche

Listing 7.1 – Test unitaire : filtrage des clubs validés pour la recherche

```
import { ClubsService } from './clubs.service';

describe('ClubsService - searchPublicClubs', () => {
  let service;
  const fakeRepo = { searchByQuery: jest.fn() };

  beforeEach(() => {
    service = new ClubsService(fakeRepo);
  });

  it('retourne uniquement les clubs VALIDATED correspondant à la recherche'
      , async () => {
    fakeRepo.searchByQuery.mockResolvedValueOnce([
      { id: 1, name: 'Aquaclub21', city: 'Dijon', status: 'VALIDATED' },
      { id: 2, name: 'Test Suba', city: 'Dijon', status: '
          PENDING_VALIDATION' },
    ]);

    const result = await service.searchPublicClubs('Dijon');

    expect(result).toHaveLength(1);
    expect(result[0].name).toBe('Aquaclub21');
    expect(result[0].status).toBe('VALIDATED');
  });
});
```

Exemple 2 : vérification des règles d'inscription à un événement

Listing 7.2 – Test unitaire : règles métier d'inscription à un événement

```
import { EventsService } from './events.service';

describe('EventsService - canUserRegisterToEvent', () => {
  let service;

  beforeEach(() => {
    service = new EventsService();
  });
```

```
9
10    it('retourne false si l événement est complet', () => {
11      const user = { hasValidLicense: true, hasValidMedicalCertificate: true
            };
12      const event = { capacity: 10, registeredCount: 10 };
13
14      const result = service.canUserRegisterToEvent(user, event);
15
16      expect(result).toBe(false);
17    });
18
19    it('retourne true si toutes les conditions sont remplies', () => {
20      const user = { hasValidLicense: true, hasValidMedicalCertificate: true
            };
21      const event = { capacity: 10, registeredCount: 5 };
22
23      const result = service.canUserRegisterToEvent(user, event);
24
25      expect(result).toBe(true);
26    });
27  });
```

Les tests unitaires représentent plus de 70% de la stratégie de tests du projet.

7.1.2 Tests d'intégration

Les tests d'intégration valident l'interaction entre plusieurs composants NestJS : contrôleurs, services, modules, base de données, validation, guards, etc.

Pourquoi réaliser des tests d'intégration ?

— Vérifier que l'API fonctionne réellement (pas seulement la logique interne).
— Tester l'authentification JWT et la gestion des rôles.
— Vérifier l'interaction entre NestJS, Prisma et PostgreSQL.
— Détecter des erreurs impossibles à identifier avec de simples tests unitaires.

Valeur ajoutée des tests d'intégration

— Validation réaliste des endpoints.
— Test des modules tels qu'ils seront utilisés en production.
— Couverture des dépendances entre services.

Exemple 1 : test d'intégration de la création d'un club en attente

Listing 7.3 – Test d'intégration : proposition d'un club en statut PENDING_VALIDATION

```
1  import { INestApplication } from '@nestjs/common';
2  import { Test } from '@nestjs/testing';
3  import * as request from 'supertest';
4  import { AppModule } from '../src/app.module';
5
6  describe('ClubsController (integration)', () => {
7    let app;
8
9    beforeAll(async () => {
10     const moduleRef = await Test.createTestingModule({
11       imports: [AppModule],
```

```
    }).compile();

    app = moduleRef.createNestApplication();
    await app.init();
  });

  afterAll(async () => app.close());

  it('/clubs/propose crée un club en statut PENDING_VALIDATION', async ()
      => {
    const response = await request(app.getHttpServer())
      .post('/clubs/propose')
      .send({
        name: 'Club Test CDA',
        city: 'Dijon',
        email: 'contact@club-test.fr',
      })
      .expect(201);

    expect(response.body.id).toBeDefined();
    expect(response.body.status).toBe('PENDING_VALIDATION');
  });
});
```

Exemple 2 : test d'intégration de la recherche de clubs

Listing 7.4 – Test d'intégration : recherche de clubs publics

```
import { INestApplication } from '@nestjs/common';
import { Test } from '@nestjs/testing';
import * as request from 'supertest';
import { AppModule } from '../src/app.module';

describe('Public clubs search (integration)', () => {
  let app;

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [AppModule],
    }).compile();

    app = moduleRef.createNestApplication();
    await app.init();
  });

  afterAll(async () => app.close());

  it('/clubs/search ne retourne que des clubs VALIDATED', async () => {
    const response = await request(app.getHttpServer())
      .get('/clubs/search')
      .query({ query: 'Dijon' })
      .expect(200);

    expect(Array.isArray(response.body)).toBe(true);
    response.body.forEach((club) => {
      expect(club.status).toBe('VALIDATED');
    });
  });
```

```
});
```

Ces tests couvrent environ 20% de la stratégie globale.

7.1.3 Tests End-to-End (E2E)

Les tests End-to-End valident les parcours critiques des utilisateurs à travers toute la plateforme. Ils simulent un vrai utilisateur interagissant via le navigateur (Next.js) avec le backend NestJS.

Pourquoi réaliser des tests E2E ?

— Reproduire des scénarios réels (connexion, ajout de plongée, navigation).
— S'assurer que front-end, back-end et base de données fonctionnent ensemble.
— Valider les parcours critiques avant mise en production.

Valeur ajoutée des tests E2E

— Détection des erreurs visibles par les utilisateurs.
— Validation de la cohérence globale du système.
— Réduction des bugs en production.

Exemple 1 : scénario E2E de recherche et affichage d'un club

Listing 7.5 – Test E2E : connexion et recherche d'un club

```
import { test, expect } from '@playwright/test';

test('un utilisateur peut se connecter et afficher la page d un club',
    async ({ page }) => {
  await page.goto('http://localhost:3000');

  // Connexion
  await page.getByRole('link', { name: 'Connexion' }).click();
  await page.getByLabel('Email').fill('user@test.fr');
  await page.getByLabel('Mot de passe').fill('Password123!');
  await page.getByRole('button', { name: 'Se connecter' }).click();

  await expect(page.getByText('Bienvenue')).toBeVisible();

  // Recherche d'un club
  await page.getByPlaceholder('Rechercher un club').fill('Aquaclub21');
  await page.getByRole('button', { name: 'Rechercher' }).click();

  // Accès à la page du club
  await page.getByRole('link', { name: 'Aquaclub21' }).click();
  await expect(page.getByText('Aquaclub21 - Page du club')).toBeVisible();
});
```

Exemple 2 : scénario E2E de validation d'un club et visibilité publique

Listing 7.6 – Test E2E : validation d'un club côté back office, visibilité côté front

```
import { test, expect } from '@playwright/test';

test('un admin valide un club qui devient visible dans la recherche
    publique', async ({ page }) => {
```

```
  // Connexion admin technique
  await page.goto('http://localhost:3000/admin');
  await page.getByLabel('Email').fill('admin-tech@divingoclub.fr');
  await page.getByLabel('Mot de passe').fill('AdminPassword123!');
  await page.getByRole('button', { name: 'Se connecter' }).click();

  await expect(page.getByText('Back Office - Clubs à valider')).toBeVisible
      ();

  // Validation du club en attente
  await page.getByRole('row', { name: /Club Test CDA/ }).getByRole('button'
      , { name: 'Valider' }).click();
  await expect(page.getByText('Club validé avec succès')).toBeVisible();

  // Vérification côté front public
  await page.goto('http://localhost:3000');
  await page.getByPlaceholder('Rechercher un club').fill('Club Test CDA');
  await page.getByRole('button', { name: 'Rechercher' }).click();

  await expect(page.getByRole('link', { name: 'Club Test CDA' })).
      toBeVisible();
});
```

Les tests E2E représentent environ 10% du total mais couvrent les scénarios les plus critiques.

Conclusion

La pyramide de tests mise en place pour *Diving O Club* garantit une qualité élevée, une détection rapide des erreurs et une validation solide des parcours utilisateurs. Cette stratégie est adaptée à un projet multi-tenant complexe et assure une stabilité sur le long terme.

7.2 Tests de performance

Les tests de performance ont pour objectif de garantir que l'application *Diving O Club* reste stable, rapide et réactive, même sous une charge importante. Ils permettent d'identifier les goulots d'étranglement, de valider les objectifs de latence et de s'assurer que les fonctionnalités critiques restent accessibles pour les utilisateurs des clubs, même en période de forte activité (campagnes d'adhésion, inscriptions aux événements, etc.).

Les métriques principales utilisées sont :

— **Latence P95** : 95% des requêtes doivent répondre en dessous d'un seuil défini.
— **Débit (throughput)** : nombre de requêtes traitées par seconde.
— **Taux d'erreurs** : proportion de requêtes échouées sous charge.

Ces tests sont réalisés avec l'outil **k6**, adapté aux tests de charge sur API REST et facilement intégrable dans une pipeline CI/CD.

7.2.1 Scénarios de performance testés

Les scénarios sont choisis en fonction des fonctionnalités critiques de la V1 :

— **Authentification** : pic de connexions simultanées en période d'ouverture des inscriptions.
— **Recherche de clubs** : endpoint fortement sollicité par les nouveaux utilisateurs.
— **Affichage de la page publique d'un club** : nombre élevé de consultations.
— **Validation d'un club côté Back Office** : opération sensible nécessitant une réactivité stable.

Chaque scénario est testé sous une montée progressive de charge afin d'observer :

— le comportement nominal de l'API,
— la dégradation éventuelle sous charge,
— le seuil à partir duquel l'application commence à ralentir.

7.2.2 Exemple de script de test de performance

L'exemple suivant illustre un test de montée en charge sur l'endpoint de recherche des clubs, réalisé avec `k6`. Le test vérifie notamment que 95% des requêtes répondent en moins de 300 ms.

Listing 7.7 – Script k6 : test de charge sur la recherche de clubs

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  stages: [
    { duration: '20s', target: 50 },  // montée progressive jusqu'à 50 utilisateurs
    { duration: '30s', target: 50 },  // plateau
    { duration: '10s', target: 0 },   // descente
  ],
  thresholds: {
    http_req_failed: ['rate<0.01'],     // moins de 1% d'erreurs
    http_req_duration: ['p(95)<300'],   // P95 inférieur à 300ms
  },
};

export default function () {
  const res = http.get('https://api.divingoclub.fr/clubs/search?query=Dijon');

  check(res, {
    'status 200': (r) => r.status === 200,
    'P95 OK':     (r) => r.timings.duration < 300,
  });

  sleep(1);
}
```

7.2.3 Interprétation des résultats

À la suite de l'exécution du script, plusieurs métriques essentielles sont analysées :

— **P95 stable** : si le P95 reste en dessous de l'objectif (ex. 300 ms), l'expérience utilisateur est considérée comme fluide.

— **Débit constant** : un throughput régulier indique que la plateforme absorbe correctement la charge.

— **Taux d'erreur faible** : un taux inférieur à 1% garantit la fiabilité de l'API.

Ces mesures permettent d'anticiper les montées en charge en conditions réelles, notamment lors des ouvertures d'inscriptions, des campagnes d'adhésion ou des pics d'activité saisonniers.

7.2.4 Analyse des performances front-end avec Lighthouse

En complément des tests de charge sur l'API, l'interface web de *Diving O Club* est auditée à l'aide de **Lighthouse**, un outil automatisé intégré dans Chrome. Lighthouse évalue plusieurs dimensions essentielles à l'expérience utilisateur :

— **Performance** : temps de chargement, interactivité, Largest Contentful Paint (LCP).

— **Accessibilité** : contraste, labels de formulaires, navigabilité clavier.

— **SEO** : conformité aux bonnes pratiques de référencement.

— **Best Practices** : sécurité front-end, gestion correcte des ressources.

Les pages testées incluent :

— la page de recherche de clubs ;
— la page publique d'un club ;
— la page d'authentification.

Les objectifs définis pour la V1 sont :

— **Performance > 80** ;
— **Accessibilité > 90** ;
— **LCP < 2,5 secondes**.

Lighthouse permet ainsi de garantir que l'application reste rapide et agréable, y compris sur mobile ou sur des connexions instables.

7.2.5 Tests d'accessibilité avec WAVE

L'accessibilité est également vérifiée à l'aide de **WAVE (Web Accessibility Evaluation Tools)**, un outil spécialisé dans la détection des barrières rencontrées par les personnes présentant des handicaps.

WAVE analyse notamment :

— la présence de textes alternatifs sur les images ;
— la bonne utilisation de la structure sémantique (titres, sections) ;
— les contrastes et la lisibilité ;
— la navigation au clavier ;
— l'absence d'erreurs ARIA ;
— la cohérence des champs de formulaires.

L'objectif est de garantir une application conforme aux recommandations **WCAG 2.1** et accessible à tous les membres et visiteurs des clubs de plongée.

7.3 Qualité du code avec SonarQube

La qualité du code est un élément fondamental dans la pérennité d'un projet à long terme comme *Diving O Club*, qui doit rester maintenable, compréhensible et évolutif. Pour répondre à ces exigences, l'analyse statique du code est réalisée à l'aide de **SonarQube** (ou SonarCloud dans le cadre de l'intégration GitHub).

SonarQube permet de détecter automatiquement les anomalies dans le code source, de mesurer la qualité globale du projet et de suivre son évolution au fil des versions. L'objectif est d'éviter l'accumulation de dette technique, d'améliorer la lisibilité du code et de réduire les risques de bugs en production.

7.3.1 Objectifs de l'analyse qualité

L'utilisation de SonarQube poursuit plusieurs objectifs complémentaires :

— **Détection des bugs potentiels** : erreurs de logique, exceptions possibles, mauvaises pratiques.
— **Détection des vulnérabilités** : injections, mauvaise gestion de l'authentification, données sensibles.
— **Réduction des code smells** : complexité excessive, duplications, non-respect des standards.
— **Suivi de la couverture de tests** : vérifier que les tests unitaires et d'intégration couvrent suffisamment le code.
— **Gestion de la dette technique** : estimation du temps nécessaire pour corriger les problèmes détectés.

Ces analyses permettent d'assurer une base de code saine, facilitant le travail du développeur et la montée en compétence des futurs contributeurs.

7.3.2 Métriques surveillées

SonarQube fournit un ensemble d'indicateurs permettant d'évaluer objectivement la qualité du code :

— **Coverage** : pourcentage du code couvert par les tests automatisés.
— **Bugs** : erreurs pouvant entraîner un comportement incorrect.
— **Vulnérabilités** : failles de sécurité potentielles.
— **Code Smells** : mauvaises pratiques impactant la maintenabilité.
— **Duplication** : proportion de lignes de code dupliquées.
— **Debt Ratio** : temps estimé pour corriger les problèmes détectés.

Pour la V1 de *Diving O Club*, les objectifs qualitatifs sont les suivants :

— **Couverture des tests supérieure à 70%** ;
— **Aucune vulnérabilité critique ou majeure** ;
— **Taux de duplication inférieur à 3%**.

7.3.3 Intégration dans la CI/CD

SonarQube est intégré directement dans la chaîne de développement via GitHub Actions. À chaque pull request vers la branche `develop`, la pipeline effectue :

1. l'installation des dépendances ;
2. l'exécution des tests unitaires et d'intégration (générant un rapport de couverture) ;
3. l'analyse du code par SonarQube ;
4. la publication d'un rapport détaillé dans l'interface SonarCloud.

Si des problèmes critiques sont détectés (vulnérabilités, duplication importante, couverture insuffisante), la pull request peut être bloquée, garantissant ainsi un niveau de qualité homogène.

7.3.4 Exemple de pipeline CI GitHub

L'extrait suivant illustre une configuration simplifiée pour exécuter une analyse SonarCloud :

Listing 7.8 – Pipeline CI : analyse SonarCloud

```yaml
name: CI Quality

on:
  pull_request:
    branches: [ "develop" ]

jobs:
  sonar:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Install Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 18

      - name: Install dependencies
        run: npm install

```

```
      - name: Run tests with coverage
        run: npm run test:coverage

      - name: SonarCloud Scan
        uses: SonarSource/sonarcloud-github-action@master
        env:
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        with:
          projectKey: diving-o-club
          organization: divingoclub
```

7.3.5 Conclusion

Grâce à SonarQube, la qualité du code de *Diving O Club* est contrôlée en continu, permettant :

— une réduction progressive de la dette technique ;
— une amélioration de la maintenabilité ;
— une meilleure sécurité du code source ;
— une fiabilité accrue avant chaque mise en production.

Cette démarche s'inscrit dans une vision de long terme et pose les bases d'un projet robuste, évolutif et durable.

7.4 Liens utiles

— Jest Documentation : https://jestjs.io/docs/getting-started
— Cypress Testing : https://docs.cypress.io/
— SonarQube : https://docs.sonarsource.com/sonarqube/latest/
— Lighthouse CI : https://developers.google.com/web/tools/lighthouse-ci
— k6 Performance Testing : https://k6.io/docs/
— Testing Best Practices : https://testingjavascript.com/

Chapitre 8

Déploiement et CI/CD

8.1 Containerisation avec Docker

La containerisation Docker standardise l'environnement de développement et de production, garantissant la reproductibilité des déploiements. Le Dockerfile multi-stage optimise la taille des images en séparant les phases de build et de runtime. Docker Compose orchestre les services (application, base de données, cache) pour un environnement complet.

Les images Docker sont optimisées pour la sécurité avec des utilisateurs non-root et des images de base minimales. Le cache des layers Docker accélère les builds et réduit la consommation de bande passante.

Exemple

Dockerfile multi-stage :

```
\# Stage 1: Build
FROM node:18-alpine AS builder

WORKDIR /app

\# Copier les fichiers de dépendances
COPY package*.json ./
RUN npm ci --only=production

\# Copier le code source
COPY . .

\# Build de l'application
RUN npm run build

\# Stage 2: Production
FROM node:18-alpine AS production

\# Créer un utilisateur non-root
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nextjs -u 1001

WORKDIR /app

\# Copier les dépendances de production
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package*.json ./

\# Changer le propriétaire des fichiers
RUN chown -R nextjs:nodejs /app
USER nextjs

\# Exposer le port
EXPOSE 3000

\# Variables d'environnement
ENV NODE_ENV=production
ENV PORT=3000

\# Commande de démarrage
CMD ["node", "dist/index.js"]
```

Docker Compose pour l'environnement complet (1/2) :

```
version: '3.8'

services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - DATABASE_URL=postgresql://user:pass@postgres:5432/projectdb
      - MONGODB_URI=mongodb://mongo:27017/projectlogs
    depends_on:
      - postgres
      - mongo
      - redis
    restart: unless-stopped

```

Exemple

Docker Compose pour l'environnement complet (2/2) :

```
  mongo:
    image: mongo:6
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
    volumes:
      - mongo_data:/data/db
    ports:
      - "27017:27017"
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    restart: unless-stopped

volumes:
  postgres_data:
  mongo_data:
```

À FAIRE / À VÉRIFIER

- Utiliser des Dockerfiles multi-stage pour optimiser les images
- Créer des utilisateurs non-root pour la sécurité
- Organiser les services avec Docker Compose
- Optimiser le cache des layers Docker
- Surveiller la taille et la sécurité des images

Contrôles Jury CDA

- Pourquoi utiliser Docker pour votre application ?
- Comment optimisez-vous vos images Docker ?
- Votre Dockerfile est-il sécurisé ?
- Comment gérez-vous les secrets dans Docker ?
- Avez-vous testé vos conteneurs en production ?

8.2 Pipeline CI/CD avec GitHub Actions

Le pipeline CI/CD automatise les étapes de linting, build, test, scan de sécurité et déploiement. GitHub Actions exécute ces étapes à chaque push et Pull Request, garantissant la qualité du code avant intégration. Les secrets et variables d'environnement sécurisent les informations sensibles.

Le déploiement automatique vers les environnements de staging et production suit une approche blue-green pour minimiser les risques. Les rollbacks automatiques sont déclenchés en cas de détection d'anomalies.

Exemple

Workflow GitHub Actions complet (1/3) :

```yaml
name: CI/CD Pipeline

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main, develop]

env:
  NODE_VERSION: '18'
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  # Job 1: Lint et tests
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ env.NODE_VERSION }}
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run linter
        run: npm run lint

      - name: Run type checking
        run: npm run type-check

      - name: Run tests
        run: npm test -- --coverage

      - name: Upload coverage
        uses: codecov/codecov-action@v3
        with:
          token: ${{ secrets.CODECOV_TOKEN }}
```

Exemple

Workflow GitHub Actions complet (2/3) :

```yaml
  # Job 2: Build et scan de sécurité
  build-and-scan:
    runs-on: ubuntu-latest
    needs: test
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Build Docker image
        run: docker build -t ${{ env.IMAGE_NAME }}:${{ github.sha }} .

      - name: Run Trivy security scan
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: ${{ env.IMAGE_NAME }}:${{ github.sha }}
          format: 'sarif'
          output: 'trivy-results.sarif'

      - name: Upload Trivy scan results
        uses: github/codeql-action/upload-sarif@v2
        with:
          sarif_file: 'trivy-results.sarif'
```

Exemple

Workflow GitHub Actions complet (3/3) :

```yaml
  # Job 3: Déploiement staging
  deploy-staging:
    runs-on: ubuntu-latest
    needs: build-and-scan
    if: github.ref == 'refs/heads/develop'
    environment: staging
    steps:
      - name: Deploy to staging
        run: |
          echo "Deploying to staging environment"
          # Script de déploiement staging
          ./scripts/deploy.sh staging

  # Job 4: Déploiement production
  deploy-production:
    runs-on: ubuntu-latest
    needs: build-and-scan
    if: github.ref == 'refs/heads/main'
    environment: production
    steps:
      - name: Deploy to production
        run: |
          echo "Deploying to production environment"
          # Script de déploiement production
          ./scripts/deploy.sh production

      - name: Run smoke tests
        run: |
          echo "Running smoke tests"
          npm run test:smoke

      - name: Notify team
        if: always()
        uses: 8398a7/action-slack@v3
        with:
          status: ${{ job.status }}
          channel: '\#deployments'
          webhook_url: ${{ secrets.SLACK_WEBHOOK }}
```

Exemple

Script de déploiement (1/2) :

```
#!/bin/bash
# scripts/deploy.sh

set -e

ENVIRONMENT=$1
IMAGE_TAG=${2:-latest}

echo "Deploying to $ENVIRONMENT environment with tag $IMAGE_TAG"

# Mise à jour des images Docker
docker-compose -f docker-compose.$ENVIRONMENT.yml pull
```

Exemple

Script de déploiement (2/2) :

```
# Déploiement blue-green
if [ "$ENVIRONMENT" = "production" ]; then
    # Déploiement en blue-green
    docker-compose -f docker-compose.prod.yml up -d --scale app=2
    sleep 30
    docker-compose -f docker-compose.prod.yml up -d --scale app=1
else
    # Déploiement simple pour staging
    docker-compose -f docker-compose.staging.yml up -d
fi

# Vérification de santé
echo "Checking application health..."
curl -f http://localhost:3000/health || exit 1

echo "Deployment to $ENVIRONMENT completed successfully"
```

Focus GitHub

Pipeline CI/CD GitHub Actions :
- **Lint :** ESLint, Prettier, TypeScript
- **Tests :** Unit, Integration, E2E
- **Sécurité :** Trivy, CodeQL, Snyk
- **Build :** Docker multi-stage
- **Deploy :** Blue-green, rollback auto

Environnements et secrets :
- **Staging :** Auto-deploy depuis develop
- **Production :** Auto-deploy depuis main
- **Secrets :** DATABASE_URL, JWT_SECRET, API_KEYS
- **Variables :** NODE_ENV, PORT, LOG_LEVEL

Métriques de pipeline :
- **Durée moyenne :** 8 minutes
- **Taux de succès :** 95%
- **Temps de déploiement :** 3 minutes
- **Rollbacks :** 2% des déploiements

À FAIRE / À VÉRIFIER

- Automatiser tous les aspects du pipeline CI/CD
- Séparer les environnements de staging et production
- Implémenter des tests de non-régression automatisés
- Configurer des alertes en cas d'échec de déploiement
- Documenter les procédures de rollback

Contrôles Jury CDA

- Votre pipeline CI/CD est-il complet ?
- Comment gérez-vous les secrets et variables ?
- Avez-vous prévu les rollbacks automatiques ?
- Comment testez-vous vos déploiements ?
- Votre pipeline respecte-t-il les bonnes pratiques ?

8.3 Documentation et monitoring

La documentation technique couvre l'API avec Swagger/OpenAPI, les procédures opérationnelles dans un runbook, et le monitoring avec des dashboards temps réel. Les logs structurés facilitent le debugging et l'analyse des performances. Les alertes automatiques notifient l'équipe en cas d'anomalie.

Le monitoring couvre les métriques applicatives (latence, débit, erreurs) et infrastructure (CPU, mémoire, disque). Les dashboards Grafana visualisent ces métriques pour faciliter la surveillance et l'analyse des tendances.

Exemple

Documentation API Swagger :

```yaml
openapi: 3.0.0
info:
  title: Project Management API
  version: 1.0.0

paths:
  /projects:
    get:
      summary: Liste des projets
      parameters:
        - name: page
          in: query
          schema:
            type: integer
            default: 1
      responses:
        '200':
          description: Liste des projets
          content:
            application/json:
              schema:
                type: object
                properties:
                  data:
                    type: array
                    items:
                      $ref: '\#/components/schemas/Project'
    post:
      summary: Créer un projet
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '\#/components/schemas/ProjectInput'
      responses:
        '201':
          description: Projet créé

components:
  schemas:
    Project:
      type: object
      properties:
        id: { type: string, format: uuid }
        name: { type: string }
        description: { type: string }
        createdAt: { type: string, format: date-time }
    ProjectInput:
      type: object
      required: [name]
      properties:
        name: { type: string, minLength: 1 }
        description: { type: string }
```

Exemple

Runbook opérationnel (1/3) :

````
# Runbook - Project Management Application

## Procédures de démarrage

### Démarrage de l'application
```bash
# Environnement de développement
docker-compose up -d

# Environnement de production
docker-compose -f docker-compose.prod.yml up -d
```

### Vérification de santé
```bash
curl -f http://localhost:3000/health
```
````

Exemple

Runbook opérationnel (2/3) :

````
## Procédures de maintenance

### Sauvegarde des données
```bash
# PostgreSQL
pg_dump -h localhost -U user projectdb > backup_$(date +%Y%m%d).sql

# MongoDB
mongodump --host localhost:27017 --db projectlogs --out backup_mongo_$(
    date +%Y%m%d)
```

### Mise à jour de l'application
```bash
# Pull de la nouvelle image
docker-compose pull

# Redémarrage avec la nouvelle image
docker-compose up -d
```
````

Exemple

Configuration de monitoring :

```
\# docker-compose.monitoring.yml
version: '3.8'

services:
  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'

  grafana:
    image: grafana/grafana
    ports:
      - "3001:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    volumes:
      - grafana_data:/var/lib/grafana

  node-exporter:
    image: prom/node-exporter
    ports:
      - "9100:9100"
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro

volumes:
  grafana_data:
```

À FAIRE / À VÉRIFIER

- Documenter l'API avec OpenAPI/Swagger
- Créer un runbook opérationnel complet
- Implémenter un monitoring proactif
- Configurer des alertes automatiques
- Former l'équipe aux procédures opérationnelles

Contrôles Jury CDA

- Votre API est-elle documentée ?
- Avez-vous un runbook opérationnel ?
- Comment surveillez-vous votre application ?
- Vos alertes sont-elles configurées ?
- L'équipe connaît-elle les procédures d'urgence ?

8.4 Liens utiles

— Dockerfile reference : https://docs.docker.com/reference/dockerfile/
— Docker Compose : https://docs.docker.com/compose/
— GitHub Actions : https://docs.github.com/actions
— Postman : https://learning.postman.com/docs/getting-started/introduction/
— Prometheus : https://prometheus.io/docs/

Chapitre 9

Veille technologique et sécurité

9.1 Veille technologique stack

La veille technologique couvre l'évolution des technologies utilisées dans le projet : React, Node.js, PostgreSQL, MongoDB, et Docker. Les sources d'information incluent les blogs officiels, GitHub releases, et les communautés techniques. Cette veille permet d'anticiper les évolutions et de planifier les mises à jour.

L'analyse des tendances technologiques guide les choix d'architecture et d'implémentation. La participation aux communautés open source et aux conférences enrichit la compréhension des bonnes pratiques et des innovations.

Exemple

Sources de veille technologique :

- **Frontend :** React Blog, Next.js Releases, TypeScript Roadmap
- **Backend :** Node.js Releases, Express.js Updates, Prisma Changelog
- **Bases de données :** PostgreSQL Release Notes, MongoDB Updates
- **DevOps :** Docker Blog, Kubernetes Releases, GitHub Actions Updates
- **Sécurité :** OWASP News, CVE Database, Security Advisories

Exemple de veille React :

```
React 18.2.0 (Janvier 2024)
+-- Nouvelles fonctionnalités
|   +-- Concurrent Features stabilisées
|   +-- Suspense amélioré
|   +-- Server Components en production
+-- Performances
|   +-- Réduction de 15% du bundle size
|   +-- Amélioration du rendu concurrent
+-- Migration
    +-- Breaking changes mineurs
    +-- Guide de migration disponible
```

Impact sur le projet :

- **React 18 :** Migration planifiée pour Q2 2024
- **Node.js 20 :** Mise à jour pour les performances
- **PostgreSQL 16 :** Nouvelles fonctionnalités JSON
- **Docker Compose V2 :** Amélioration des performances

À FAIRE / À VÉRIFIER

- Suivre les releases officielles des technologies utilisées
- Participer aux communautés techniques (GitHub, Stack Overflow)
- S'abonner aux newsletters et blogs spécialisés
- Tester les nouvelles versions en environnement de développement
- Documenter les impacts et planifier les migrations

Contrôles Jury CDA

- Quelles sources utilisez-vous pour votre veille ?
- Comment identifiez-vous les technologies émergentes ?
- Avez-vous planifié des mises à jour technologiques ?
- Comment évaluez-vous l'impact des nouvelles versions ?
- Votre veille influence-t-elle vos choix techniques ?

9.2 Bonnes pratiques sécurité

La veille sécurité suit les recommandations OWASP, les CVE (Common Vulnerabilities and Exposures), et les advisories des éditeurs. L'analyse des menaces émergentes guide l'évolution des mesures de protection. Les tests de pénétration réguliers valident l'efficacité des contrôles de sécurité.

L'application des bonnes pratiques sécurité inclut la mise à jour régulière des dépendances, la configuration sécurisée des services, et la formation de l'équipe aux risques. La documentation des incidents et des contre-mesures enrichit la base de connaissances sécurité.

Exemple

Veille sécurité OWASP 2024 :

- **A01 - Broken Access Control :** Nouveaux patterns d'attaque
- **A02 - Cryptographic Failures :** Vulnérabilités des algorithmes
- **A03 - Injection :** Évolution des techniques d'injection
- **A04 - Insecure Design :** Risques de conception
- **A05 - Security Misconfiguration :** Configurations par défaut

Exemple de vulnérabilité suivie :

```
CVE-2024-1234: Vulnerability in Express.js
+-- Severity: HIGH (CVSS 7.5)
+-- Description: Prototype pollution in req.query
+-- Affected versions: < 4.18.3
+-- Impact: Remote code execution possible
+-- Mitigation: Update to Express 4.18.3+
+-- Status: Fixed in project (v4.18.5)
```

Mesures de sécurité appliquées :

- **Dépendances :** Audit automatique avec npm audit
- **Conteneurs :** Scan de vulnérabilités avec Trivy
- **Code :** Analyse statique avec SonarQube
- **Runtime :** Monitoring des anomalies avec Prometheus
- **Formation :** Sessions sécurité trimestrielles

À FAIRE / À VÉRIFIER

- Surveiller les CVE et advisories de sécurité
- Automatiser l'audit des dépendances
- Implémenter des tests de sécurité automatisés
- Former l'équipe aux bonnes pratiques sécurité
- Documenter les incidents et les contre-mesures

Contrôles Jury CDA

- Comment surveillez-vous les vulnérabilités ?
- Avez-vous automatisé l'audit de sécurité ?
- Comment gérez-vous les vulnérabilités critiques ?
- L'équipe est-elle formée à la sécurité ?
- Avez-vous un plan de réponse aux incidents ?

9.3 Application au projet

La veille technologique et sécurité influence directement les choix d'architecture et d'implémentation du projet. Les nouvelles fonctionnalités sont évaluées selon leur impact sur la sécurité, les performances, et la maintenabilité. Les mises à jour sont planifiées selon un calendrier de migration structuré.

L'intégration des bonnes pratiques découvertes améliore continuellement la qualité du code et la sécurité de l'application. La documentation des décisions techniques facilite la transmission des connaissances et la maintenance future.

Exemple

Évolution technique du projet :

— **Q1 2024 :** Migration vers React 18 pour les performances

— **Q2 2024 :** Implémentation des Server Components

— **Q3 2024 :** Mise à jour PostgreSQL 16 pour les JSON

— **Q4 2024 :** Migration vers Node.js 20 LTS

Améliorations sécurité appliquées :

```
// AVANT : Validation basique
const validateUser = (userData) => {
  if (userData.email && userData.password) {
    return true;
  }
  return false;
};

// APRÈS : Validation robuste avec sanitisation
const validateUser = (userData) => {
  const schema = Joi.object({
    email: Joi.string().email().max(255).required(),
    password: Joi.string().min(8).pattern(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d
       )/).required(),
    name: Joi.string().max(100).sanitize().required()
  });

  const { error, value } = schema.validate(userData);
  if (error) {
    throw new ValidationError(error.details[0].message);
  }

  return value;
};
```

Métriques d'amélioration :

Aspect	Avant	Après	Amélioration
Temps de réponse	800ms	450ms	-44%
Vulnérabilités	12	0	-100%
Couverture tests	65%	85%	+31%
Bundle size	2.1MB	1.4MB	-33%

À FAIRE / À VÉRIFIER

— Intégrer les bonnes pratiques découvertes en veille

— Planifier les migrations technologiques

— Mesurer l'impact des améliorations

— Documenter les décisions techniques

— Partager les connaissances avec l'équipe

Contrôles Jury CDA

- Comment appliquez-vous votre veille au projet ?
- Avez-vous mesuré l'impact des améliorations ?
- Vos décisions techniques sont-elles documentées ?
- Comment partagez-vous vos connaissances ?
- Votre veille influence-t-elle la roadmap ?

9.4 Liens utiles

- InfoQ : https://www.infoq.com/
- OWASP News : https://owasp.org/news/
- PostgreSQL Release Notes : https://www.postgresql.org/docs/release/
- React Blog : https://react.dev/blog
- Node.js Releases : https://nodejs.org/en/about/releases/

Chapitre 10

Bilan et retour d'expérience (REX)

10.1 Objectifs atteints et non atteints

L'analyse des objectifs initiaux révèle un taux d'atteinte de 85% des objectifs SMART définis. Les objectifs métier ont été largement atteints avec la livraison du MVP dans les délais. Les objectifs techniques ont été partiellement atteints, avec quelques ajustements nécessaires pour optimiser les performances. Les objectifs pédagogiques ont été dépassés grâce aux apprentissages supplémentaires acquis.

Les objectifs non atteints concernent principalement des fonctionnalités avancées reportées en v2.0 pour respecter les contraintes temporelles. Cette priorisation a permis de livrer un produit fonctionnel et stable dans les délais impartis.

Exemple

Bilan des objectifs SMART :

Objectif	Statut	Mesure	Commentaire
Réduction temps reporting	Atteint	-42%	Dépassé l'objectif de -40%
Livraison MVP 6 mois	Atteint	5.5 mois	Livré en avance
Adoption utilisateurs	Partiel	78%	Objectif 90%, formation nécessaire
Performance P95 < 500ms	Atteint	320ms	Dépassé l'objectif
Sécurité 0 vulnérabilité	Atteint	0	Objectif atteint

Objectifs non atteints :

- **Analytics avancées :** Reporté en v2.0 (complexité technique)
- **Intégrations externes :** Reporté en v2.0 (priorités métier)
- **Mobile native :** Reporté en v2.0 (PWA suffisant)
- **IA prédictive :** Reporté en v2.0 (ROI incertain)

À FAIRE / À VÉRIFIER

- Analyser objectivement l'atteinte des objectifs
- Identifier les causes des non-atteintes
- Documenter les ajustements nécessaires
- Prévoir les actions correctives pour v2.0
- Communiquer les résultats aux parties prenantes

Contrôles Jury CDA

- Quels objectifs avez-vous atteints ?
- Pourquoi certains objectifs n'ont-ils pas été atteints ?
- Comment mesurez-vous le succès de votre projet ?
- Avez-vous ajusté vos objectifs en cours de projet ?
- Quels sont vos objectifs pour la v2.0 ?

10.2 Difficultés rencontrées et solutions

Les principales difficultés ont concerné l'intégration des bases de données hétérogènes, la gestion des performances sous charge, et la coordination des équipes distribuées. Chaque

difficulté a été analysée pour identifier les causes racines et implémenter des solutions durables.

L'approche de résolution de problèmes a combiné l'analyse technique, la recherche de solutions existantes, et l'innovation pour des cas spécifiques. La documentation des solutions facilite la réutilisation et l'amélioration continue.

Exemple

Tableau risques ☞mitigation ☞résultat :

Risque	Mitigation	Résultat	Apprentissage
Performance DB	Index + cache Redis	Latence -60%	Cache stratégique
Intégration équipes	Daily standups	Communication +40%	Processus agile
Sécurité données	Chiffrement + audit	0 incident	Sécurité by design
Délais serrés	MVP + priorités	Livraison à temps	Focus sur l'essentiel
Complexité technique	Architecture simple	Maintenance facile	KISS principle

Exemple de difficulté résolue :

```
Problème: Latence élevée des requêtes PostgreSQL
+-- Symptômes
|   +-- Temps de réponse > 2s
|   +-- Timeout des requêtes complexes
|   +-- Surcharge CPU base de données
+-- Analyse
|   +-- Requêtes sans index appropriés
|   +-- Jointures sur de gros volumes
|   +-- Pas de cache applicatif
+-- Solutions implémentées
|   +-- Création d'index composites
|   +-- Optimisation des requêtes
|   +-- Mise en place de Redis cache
|   +-- Pagination des résultats
+-- Résultat
    +-- Latence réduite à 200ms
    +-- CPU base stabilisé
    +-- Expérience utilisateur améliorée
```

À FAIRE / À VÉRIFIER

- Documenter toutes les difficultés rencontrées
- Analyser les causes racines des problèmes
- Rechercher des solutions existantes avant d'innover
- Tester les solutions avant déploiement
- Partager les apprentissages avec l'équipe

Contrôles Jury CDA

- Quelles ont été vos principales difficultés ?
- Comment avez-vous résolu ces difficultés ?
- Avez-vous documenté vos solutions ?
- Ces difficultés étaient-elles prévisibles ?
- Comment éviterez-vous ces difficultés à l'avenir ?

10.3 Dettes techniques et apprentissages

Les dettes techniques identifiées incluent la refactorisation de certains composants React, l'optimisation des requêtes MongoDB, et l'amélioration de la couverture de tests. Ces dettes sont documentées avec des priorités et des estimations pour faciliter la planification des futures itérations.

Les apprentissages techniques couvrent l'architecture microservices, la gestion des performances, et les bonnes pratiques de sécurité. Ces connaissances sont transférables à d'autres projets et enrichissent l'expertise de l'équipe.

Exemple

Registre des dettes techniques :

Dette	Priorité	Effort	Impact	Planification
Refactor composants React	Moyenne	2 semaines	Maintenabilité	v1.2
Optimisation requêtes Mongo	Haute	1 semaine	Performance	v1.1
Tests E2E manquants	Haute	1 semaine	Qualité	v1.1
Documentation API	Basse	3 jours	Développement	v1.3
Migration TypeScript	Moyenne	3 semaines	Robustesse	v2.0

Apprentissages transférables :

— **Architecture :** Pattern Repository pour l'abstraction des données

— **Performance :** Stratégies de cache multi-niveaux

— **Sécurité :** Implémentation JWT avec refresh tokens

— **Tests :** Pyramide de tests avec couverture optimale

— **DevOps :** Pipeline CI/CD avec déploiement blue-green

Exemple d'apprentissage concret :

```
// AVANT : Gestion d'état complexe
const [projects, setProjects] = useState([]);
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);

// APRÈS : Hook personnalisé réutilisable
const useProjects = () => {
  const [state, setState] = useState({
    data: [],
    loading: false,
    error: null
  });

  const fetchProjects = useCallback(async () => {
    setState(prev => ({ ...prev, loading: true }));
    try {
      const projects = await projectService.getAll();
      setState({ data: projects, loading: false, error: null });
    } catch (err) {
      setState(prev => ({ ...prev, loading: false, error: err.message }))
          ;
    }
  }, []);

  return { ...state, fetchProjects };
};
```

À FAIRE / À VÉRIFIER

- Identifier et documenter toutes les dettes techniques
- Prioriser les dettes selon leur impact et urgence
- Planifier la résolution des dettes dans les futures versions
- Capitaliser sur les apprentissages pour les futurs projets
- Partager les bonnes pratiques avec l'équipe

Contrôles Jury CDA

- Quelles dettes techniques avez-vous identifiées ?
- Comment priorisez-vous ces dettes ?
- Quels apprentissages tirez-vous de ce projet ?
- Ces apprentissages sont-ils transférables ?
- Comment capitalisez-vous sur ces expériences ?

10.4 Liens utiles

- Postmortems (Google SRE) : https://sre.google/sre-book/postmortem-culture/
- Technical Debt : https://martinfowler.com/bliki/TechnicalDebt.html
- Retrospectives : https://www.atlassian.com/team-playbook/plays/retrospective
- Lessons Learned : https://bit.ly/lessons-learned
- Knowledge Management : https://bit.ly/knowledge-management

Chapitre 11

Conclusion et remerciements

11.1 Synthèse du projet

Ce projet de développement d'une application de gestion de projets a permis de mettre en pratique les compétences acquises en alternance CDA dans un contexte professionnel concret. L'architecture 3 tiers avec React, Node.js, PostgreSQL et MongoDB a démontré sa robustesse et sa scalabilité. Les objectifs métier ont été largement atteints avec une réduction de 42% du temps de reporting et une adoption utilisateur de 78%.

La démarche méthodologique Agile a facilité la collaboration et l'adaptation aux besoins évolutifs. Les bonnes pratiques de développement, de sécurité et de déploiement ont été appliquées avec succès, garantissant la qualité et la fiabilité de la solution livrée.

Exemple

Chiffres clés du projet :

Métrique	Valeur	Objectif
Durée de développement	5.5 mois	6 mois
Couverture de code	85%	80%
Performance P95	320ms	500ms
Vulnérabilités sécurité	0	0
Adoption utilisateurs	78%	90%
Temps de reporting	-42%	-40%

Technologies maîtrisées :

- **Frontend :** React 18, TypeScript, Redux Toolkit
- **Backend :** Node.js, Express.js, Prisma ORM
- **Bases de données :** PostgreSQL, MongoDB, Redis
- **DevOps :** Docker, GitHub Actions, SonarQube
- **Sécurité :** JWT, Argon2, OWASP Top 10

À FAIRE / À VÉRIFIER

- Synthétiser les résultats quantitatifs et qualitatifs
- Mettre en avant les compétences développées
- Identifier les points forts et les axes d'amélioration
- Préparer la présentation des résultats au jury
- Documenter les apprentissages pour la suite du parcours

Contrôles Jury CDA

- Pouvez-vous résumer les résultats de votre projet ?
- Quelles compétences avez-vous développées ?
- Quels sont vos points forts et faibles ?
- Comment évaluez-vous votre progression ?
- Quels sont vos objectifs pour la suite ?

11.2 Perspectives d'évolution

Les perspectives d'évolution du projet incluent le développement de la v2.0 avec des fonctionnalités avancées : analytics prédictives, intégrations externes, et intelligence artificielle. L'architecture actuelle permet une évolution progressive sans refactoring majeur. La roadmap technique prévoit la migration vers des technologies émergentes et l'optimisation continue des performances.

L'expérience acquise sur ce projet constitue une base solide pour aborder des projets plus complexes et des responsabilités techniques élargies. Les compétences développées sont directement applicables à d'autres contextes métier et technologiques.

Exemple

Roadmap technique v2.0 :

```
Q1 2025: Fonctionnalités avancées
+-- Analytics prédictives avec machine learning
+-- Intégrations API externes (Slack, Teams)
+-- Notifications push temps réel
+-- Optimisation performances (P95 < 200ms)

Q2 2025: Intelligence artificielle
+-- Assistant IA pour la gestion de projet
+-- Recommandations automatiques
+-- Détection d'anomalies
+-- Chatbot support utilisateur

Q3 2025: Évolutions technologiques
+-- Migration vers React Server Components
+-- Mise à jour Node.js 20 LTS
+-- PostgreSQL 16 nouvelles fonctionnalités
+-- Monitoring avancé avec Grafana
```

Compétences à développer :

- **Architecture :** Microservices, Event-driven architecture
- **Cloud :** AWS/Azure, Kubernetes, Serverless
- **IA/ML :** TensorFlow, PyTorch, MLOps
- **Sécurité :** Zero Trust, DevSecOps
- **Leadership :** Architecture decision records, mentoring

À FAIRE / À VÉRIFIER

- Définir une vision claire pour l'évolution du projet
- Identifier les technologies émergentes pertinentes
- Planifier les compétences à développer
- Anticiper les besoins métier futurs
- Maintenir la veille technologique

Contrôles Jury CDA

- Quelles sont vos perspectives d'évolution ?
- Comment prévoyez-vous l'évolution technique ?
- Quelles compétences souhaitez-vous développer ?
- Comment anticipez-vous les besoins futurs ?
- Votre projet est-il évolutif ?

11.3 Remerciements

Je tiens à remercier toutes les personnes qui ont contribué à la réussite de ce projet et à mon apprentissage en alternance CDA. Ces remerciements s'adressent à l'équipe technique, aux utilisateurs métier, aux formateurs, et à tous ceux qui ont partagé leur expertise et leur temps.

L'accompagnement reçu a été déterminant dans l'acquisition des compétences techniques et méthodologiques nécessaires à la réalisation de ce projet. Ces remerciements témoignent de la reconnaissance pour l'investissement de chacun dans ma formation professionnelle.

Exemple

Remerciements personnalisés :

- **Mon tuteur entreprise :** Pour son accompagnement technique et son expertise
- **L'équipe de développement :** Pour la collaboration et le partage de connaissances
- **Les utilisateurs métier :** Pour leurs retours constructifs et leur patience
- **Les formateurs CDA :** Pour la transmission des fondamentaux techniques
- **La communauté open source :** Pour les outils et ressources mis à disposition

Apprentissages clés :

- **Collaboration :** L'importance du travail d'équipe en développement
- **Communication :** La nécessité de bien communiquer avec les parties prenantes
- **Adaptabilité :** La capacité à s'adapter aux changements et contraintes
- **Qualité :** L'exigence de qualité dans le développement logiciel
- **Veille :** L'importance de la veille technologique continue

À FAIRE / À VÉRIFIER

- Exprimer sa gratitude de manière sincère et personnalisée
- Reconnaître l'apport spécifique de chaque personne
- Mettre en avant les apprentissages tirés des interactions
- Maintenir les relations professionnelles établies
- Préparer la suite du parcours avec confiance

Contrôles Jury CDA

- Qui souhaitez-vous remercier particulièrement ?
- Quels apprentissages tirez-vous de ces interactions ?
- Comment envisagez-vous la suite de votre parcours ?
- Quelles relations professionnelles avez-vous nouées ?
- Comment comptez-vous maintenir ces relations ?

11.4 Déploiement et documentation

Dans cette section, vous devez présenter votre stratégie de déploiement et la documentation technique de votre projet. Le jury attend une compréhension claire de votre approche

opérationnelle et de la maintenabilité de votre solution.

Votre stratégie de déploiement : *[Décrivez votre approche de déploiement et de documentation]*

11.4.1 Docker

Dans cette sous-section, vous devez détailler votre approche de containerisation avec Docker. Le jury attend une explication claire de votre Dockerfile et de votre orchestration.

Votre containerisation : *[Décrivez votre Dockerfile et votre approche Docker]*

Conteneurisation

Votre Dockerfile : *[Décrivez votre Dockerfile multi-stage]*

Exemple

Dockerfile multi-stage :

```
# Stage 1: Build
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
RUN npm run build

# Stage 2: Production
FROM node:18-alpine AS production
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nextjs -u 1001
WORKDIR /app
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package*.json ./
RUN chown -R nextjs:nodejs /app
USER nextjs
EXPOSE 3000
ENV NODE_ENV=production
CMD ["node", "dist/index.js"]
```

Compose

Votre Docker Compose : *[Décrivez votre orchestration des services]*

Exemple

Docker Compose pour l'environnement complet :

```yaml
version: '3.8'
services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - DATABASE_URL=postgresql://user:pass@postgres:5432/projectdb
    depends_on:
      - postgres
      - redis
    restart: unless-stopped

  postgres:
    image: postgres:15-alpine
    environment:
      - POSTGRES_DB=projectdb
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    restart: unless-stopped

volumes:
  postgres_data:
```

11.4.2 GitHub (code source)

Dans cette sous-section, vous devez présenter votre organisation du code source sur GitHub. Le jury attend une explication claire de votre structure de repository et de vos conventions.

Votre organisation GitHub : *[Décrivez votre structure de repository et vos conventions]*

Exemple

Structure du repository :

```
project-management-app/
+-- src/                          # Code source
|   +-- frontend/                 # Application React
|   +-- backend/                  # API Node.js
|   +-- shared/                   # Code partagé
+-- docs/                         # Documentation
|   +-- api/                      # Documentation API
|   +-- deployment/               # Procédures de déploiement
|   +-- architecture/             # Documentation architecture
+-- scripts/                      # Scripts utilitaires
+-- tests/                        # Tests automatisés
+-- docker/                       # Configuration Docker
+-- .github/                      # GitHub Actions et templates
```

11.4.3 CI/CD

Dans cette sous-section, vous devez présenter votre pipeline CI/CD. Le jury attend une explication claire de votre automatisation et de vos environnements.

Votre pipeline CI/CD : *[Décrivez votre automatisation et vos environnements]*

Exemple

Pipeline CI/CD GitHub Actions :

```yaml
name: CI/CD Pipeline
on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main, develop]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '18'
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test -- --coverage

  deploy-staging:
    runs-on: ubuntu-latest
    needs: test
    if: github.ref == 'refs/heads/develop'
    steps:
      - name: Deploy to staging
        run: ./scripts/deploy.sh staging

  deploy-production:
    runs-on: ubuntu-latest
    needs: test
    if: github.ref == 'refs/heads/main'
    steps:
      - name: Deploy to production
        run: ./scripts/deploy.sh production
```

11.4.4 SonarQube

Dans cette sous-section, vous devez présenter votre approche de qualité du code avec SonarQube. Le jury attend une explication claire de vos métriques et de votre intégration.

Votre qualité du code : *[Décrivez vos métriques de qualité et votre intégration SonarQube]*

Exemple

Métriques de qualité SonarQube :

Métrique	Objectif	Actuel	Statut
Couverture de code	> 80%	85%	✓
Duplication	< 3%	1.2%	✓
Complexité cyclomatique	< 10	7.3	✓
Maintenabilité	A	A	✓
Fiabilité	A	A	✓
Sécurité	A	A	✓

11.4.5 Swagger

Dans cette sous-section, vous devez présenter votre documentation API avec Swagger. Le jury attend une explication claire de votre documentation et de son utilisation.

Votre documentation API : *[Décrivez votre documentation Swagger et son utilisation]*

Exemple

Documentation API Swagger :

```yaml
openapi: 3.0.0
info:
  title: Project Management API
  version: 1.0.0
  description: API pour la gestion des projets

paths:
  /projects:
    get:
      summary: Liste des projets
      responses:
        '200':
          description: Liste des projets
          content:
            application/json:
              schema:
                type: object
                properties:
                  data:
                    type: array
                    items:
                      $ref: '#/components/schemas/Project'

components:
  schemas:
    Project:
      type: object
      properties:
        id:
          type: string
          format: uuid
        name:
          type: string
        description:
          type: string
        createdAt:
          type: string
          format: date-time
```

À FAIRE / À VÉRIFIER

- Documenter complètement votre API avec Swagger
- Intégrer SonarQube dans votre pipeline CI/CD
- Organiser votre code source de manière claire
- Automatiser tous les aspects du déploiement
- Maintenir la documentation à jour

Contrôles Jury CDA

- Comment organisez-vous votre code source ?
- Votre pipeline CI/CD est-il complet ?
- Comment mesurez-vous la qualité de votre code ?
- Votre API est-elle documentée ?
- Comment gérez-vous les déploiements ?

11.5 Liens utiles

- Dockerfile reference : https://docs.docker.com/reference/dockerfile/
- Docker Compose : https://docs.docker.com/compose/
- GitHub Actions : https://docs.github.com/actions
- SonarQube : https://docs.sonarsource.com/sonarqube/latest/
- Swagger/OpenAPI : https://swagger.io/specification/
- CDA Formation : https://www.cda.asso.fr/
- Colint.school : https://colint.school/