cahier de charge: Reseau (multi-tenancy plateform)

ALAN TCHAPDA

October 2025

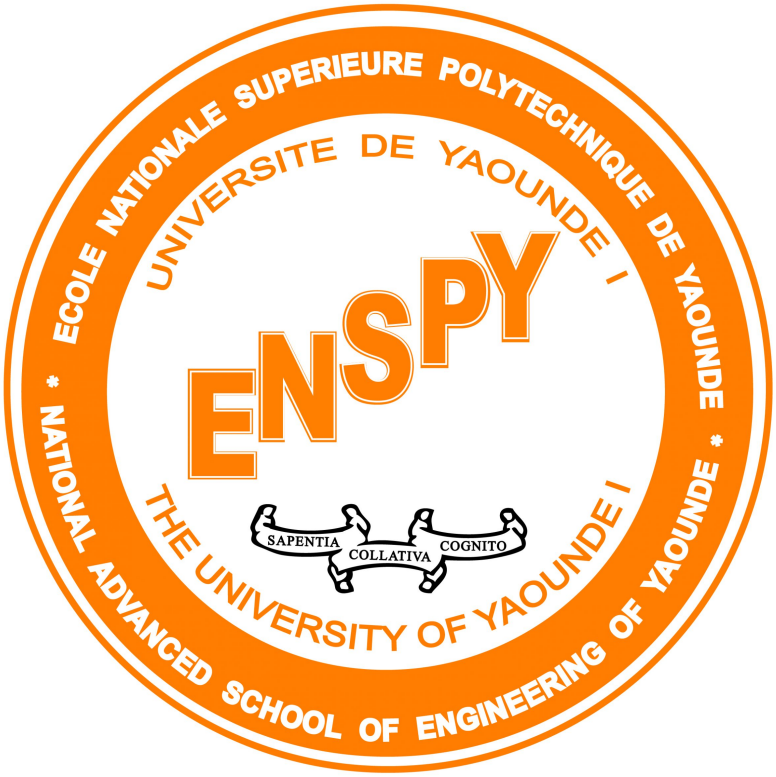

Figure 1 –

N°	Nom et prénom
1	MBOUNGANG
2	NEPESTOUM
3	WAFFO VITRIC
4	TCHAPDA ALAN
5	FUL FONKWA AMANDINE
6	FONKWA HERMANN

1 Présentation du projet

1.1 Contexte et justification

Dans un contexte où les plateformes numériques hébergent plusieurs organisations distinctes, la question de la **gestion sécurisée des utilisateurs et des accès** devient centrale. Le présent projet s'inscrit dans cette logique en proposant la conception d'une **API Spring Boot multi-tenant** inspirée du **protocole LDAP (Lightweight Directory Access Protocol)**.

L'idée principale est de permettre à plusieurs structures (ou entités) de partager une même plateforme tout en garantissant une **isolation stricte des données**. Chaque organisation doit pouvoir :

— gérer ses propres utilisateurs et groupes ;
— créer et définir ses rôles personnalisés ;
— attribuer des accès spécifiques aux utilisateurs selon leurs responsabilités ;
— assurer que les données ne soient jamais visibles ou modifiables par une autre organisation.

Cette approche favorise la **modularité**, la **scalabilité** et la **sécurité** tout en permettant une extension future vers des architectures distribuées basées sur des microservices.

1.2 Objectifs généraux

Le projet vise à développer une solution complète de gestion d'utilisateurs dans un environnement multi-tenant, en s'inspirant du modèle LDAP. Les principaux objectifs sont :

— concevoir une architecture de données hiérarchique similaire à un annuaire LDAP ;
— implémenter une API REST capable de gérer l'authentification et l'autorisation multi-tenant ;
— permettre à chaque organisation de définir dynamiquement ses rôles, groupes et permissions ;
— poser les bases d'une architecture extensible vers des microservices et intégrable avec d'autres applications.

1.3 Problématique

Comment concevoir une API de gestion d'utilisateurs multi-tenant inspirée de LDAP, garantissant à la fois :

— l'isolation complète des données entre les organisations ;
— une gestion flexible et hiérarchique des rôles et permissions ;
— une authentification sécurisée et centralisée ;
— et une ouverture vers des extensions futures (notifications, logs, monitoring, etc.) ?

1.4 Enjeux du projet

Ce projet se veut à la fois **pédagogique** et **technologique**. Il permettra aux membres de l'équipe de :

— maîtriser les concepts d'authentification, d'autorisation et d'isolation des données dans un environnement partagé ;
— comprendre la structure et les principes du protocole LDAP ;

— développer des compétences avancées en conception d'API sécurisée avec Spring Boot ;
— intégrer les bonnes pratiques d'architecture logicielle (multi-tenant, sécurité, résilience, extensibilité).

2 Description du projet

2.1 Fonctionnalités principales

Le système proposé est une **API de gestion d'utilisateurs multi-tenant** inspirée du protocole LDAP. Elle permettra à plusieurs organisations (appelées *tenants*) de coexister sur une même plateforme sans interférence entre leurs données.

Les fonctionnalités principales sont les suivantes :

1. **Gestion des organisations (tenants)** :
   — Création et administration d'une organisation.
   — Isolation complète des données entre organisations.
   — Attribution d'un *tenant_id* unique à chaque structure.
2. **Gestion des utilisateurs** :
   — Création, mise à jour, suppression et recherche d'utilisateurs.
   — Association d'un utilisateur à une ou plusieurs organisations.
   — Gestion du profil utilisateur (nom, email, mot de passe, rôle, statut).
3. **Gestion des rôles et permissions** :
   — Définition de rôles personnalisés par organisation.
   — Attribution dynamique des permissions selon les besoins internes.
   — Hiérarchisation des accès (ex. : Super Admin, Admin, Manager, Utilisateur).
4. **Authentification et autorisation** :
   — Système d'authentification inspiré de LDAP (*bind, search, access control*).
   — Génération et validation de jetons d'accès (JWT).
   — Gestion des sessions et expiration automatique des tokens.
5. **Audit et journalisation** :
   — Suivi des connexions et opérations sensibles.
   — Historique des modifications par utilisateur.
   — Export des logs à des fins de contrôle ou d'analyse.
6. **Interface de démonstration (Front-end)** :
   — Interface Next.js pour visualiser les organisations, utilisateurs et rôles.
   — Démonstration des mécanismes d'accès selon les rôles.

2.2 Besoins fonctionnels

Les besoins fonctionnels décrivent les services que le système doit offrir.

— L'administrateur global peut créer et supprimer des organisations.
— Chaque organisation peut créer, modifier ou supprimer ses propres utilisateurs.
— Chaque organisation peut définir ses rôles et leurs permissions.
— Un utilisateur ne peut accéder qu'aux ressources de son organisation dependament de son role.
— Délégation "sous couvert" : Permettre à un admin d'agir au nom d'un utilisateur avec traçabilité complète.
— Le système vérifie les identifiants d'un utilisateur avant toute opération sensible.
— Le système journalise chaque connexion et modification effectuée.
— Le front-end permet de tester les différents rôles et accès.

2.3 Besoins non fonctionnels

Les besoins non fonctionnels précisent les contraintes de qualité et de performance.

— **Sécurité** — Les mots de passe doivent être chiffrés (BCrypt ou Argon2).
— **Confidentialité** — Les données d'un tenant ne doivent jamais être accessibles à un autre.
— **Disponibilité** — Le système doit être opérationnel à 99% du temps.
— **Performance** — Les requêtes d'authentification doivent s'exécuter en moins de 300 ms.
— **Scalabilité** — L'architecture doit pouvoir évoluer vers un système distribué (microservices).
— **Maintenabilité** — Le code doit être modulaire, documenté et testé.
— **Traçabilité** — Toutes les opérations critiques doivent être enregistrées et consultables.
— **Compatibilité** — L'API doit être conforme aux normes REST et accessible depuis des applications web ou mobiles.

3 Contraintes et Exigences

3.1 Contraintes fonctionnelles

Les contraintes fonctionnelles définissent les limites opérationnelles que le système doit respecter :

- Chaque organisation (tenant) doit disposer d'un espace de données **isolé** : aucune donnée ne doit être accessible par une autre organisation.
- L'accès aux informations au sein d'une organisation doit varier selon les rôles définis.
- Chaque organisation peut créer, modifier et supprimer ses propres rôles et définir les permissions associées.
- L'API doit permettre la gestion des utilisateurs, des rôles et des groupes, avec traçabilité complète des actions.
- Les opérations critiques (authentification, délégation sous couvert, modification de rôles) doivent être journalisées.

3.2 Diagrammes fonctionnels

Les diagrammes suivants présentent les principaux cas d'utilisation et interactions entre les composants du système.

3.2.1 Diagrammes des cas d'utilisations

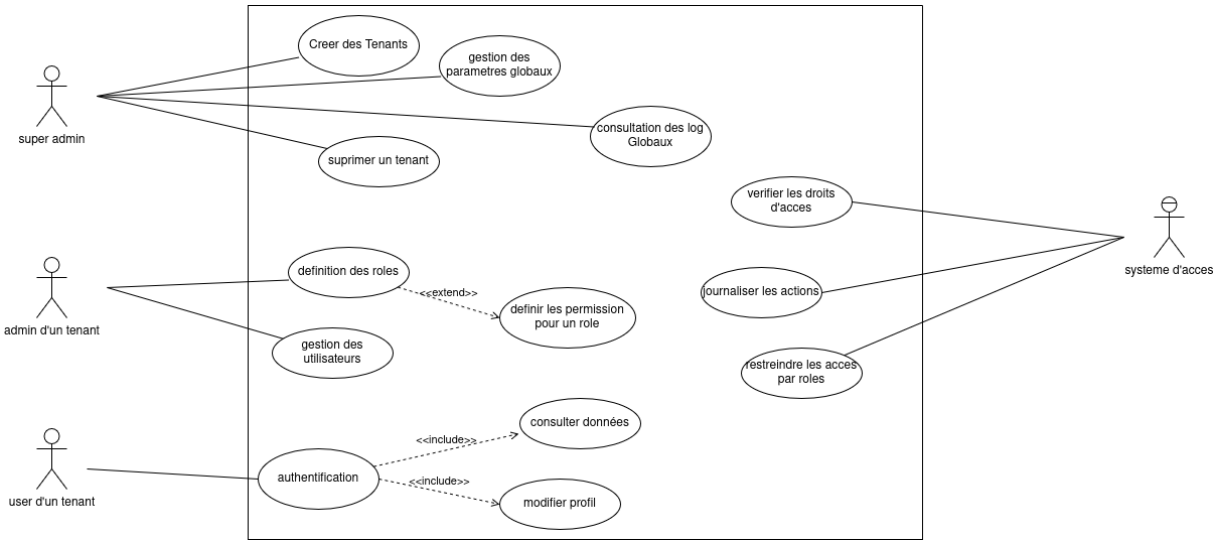

FIGURE 2 – Diagramme de cas d'utilisation principal du système

3.2.2 Diagramme de classe

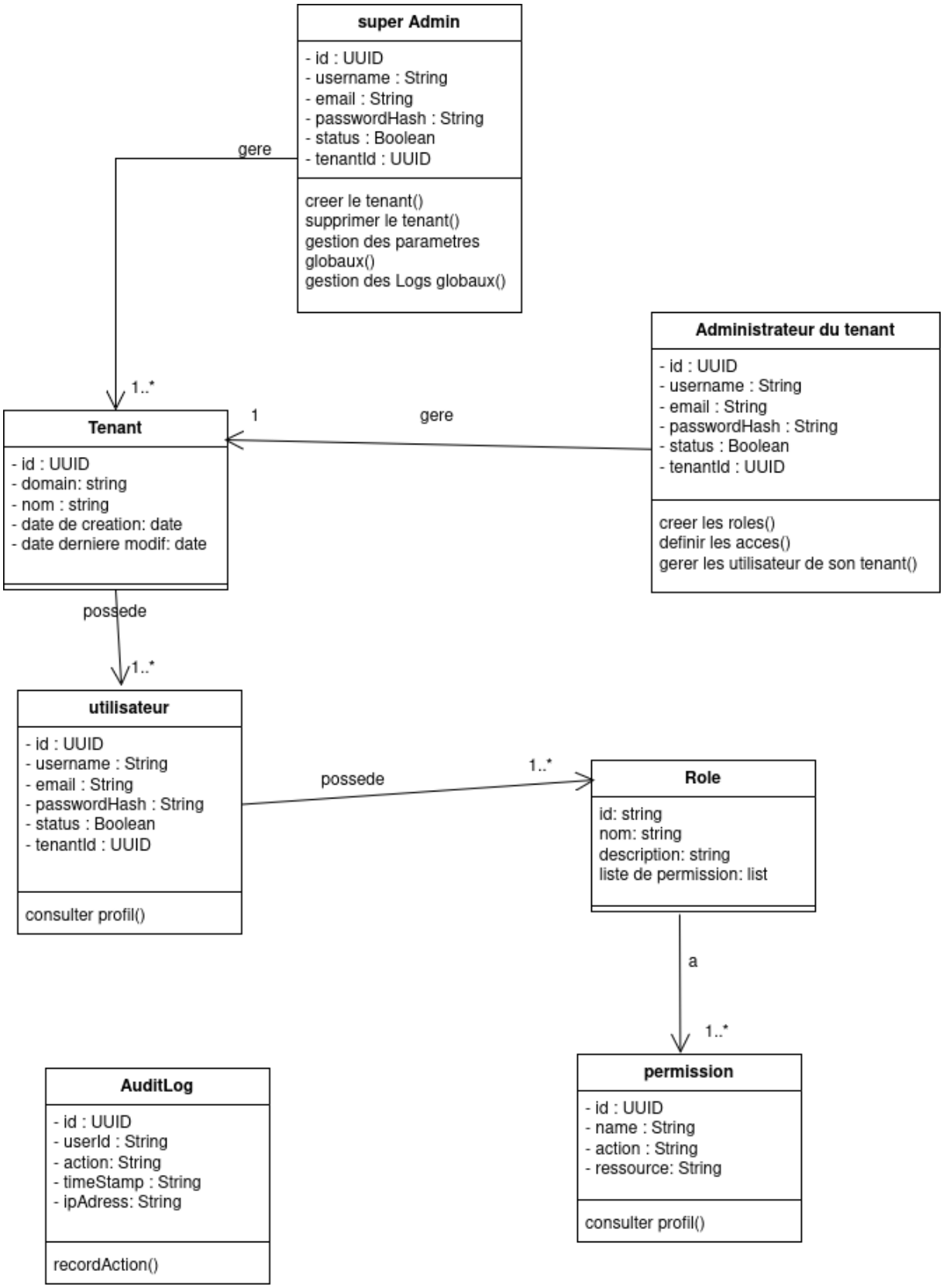

FIGURE 3 – Diagramme de classe

3.2.3 Diagramme sequence technique : Authemtification

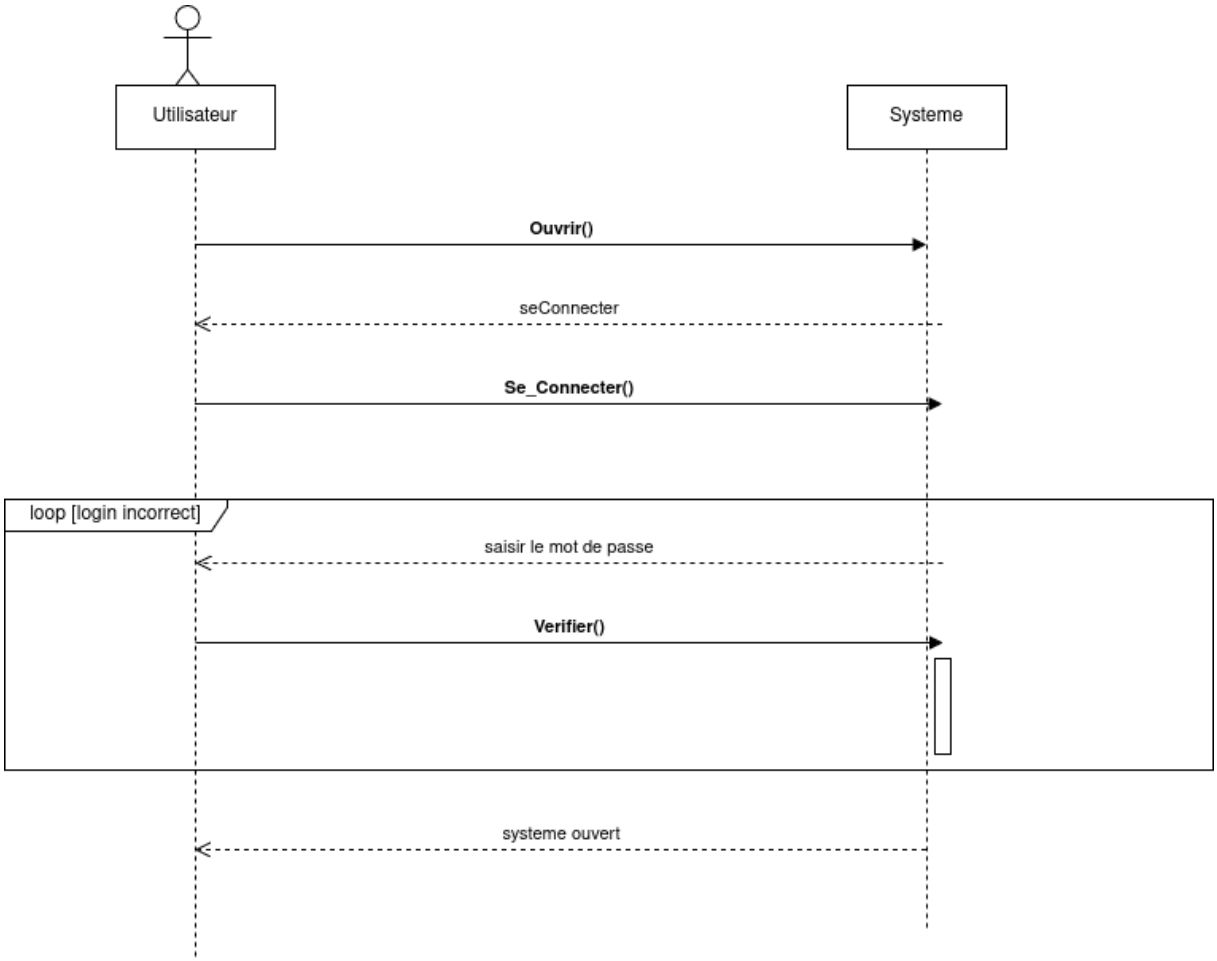

FIGURE 4 – Diagramme sequence technique : Authemtification

3.2.4 Diagramme de sequences systeme : créer un tenant

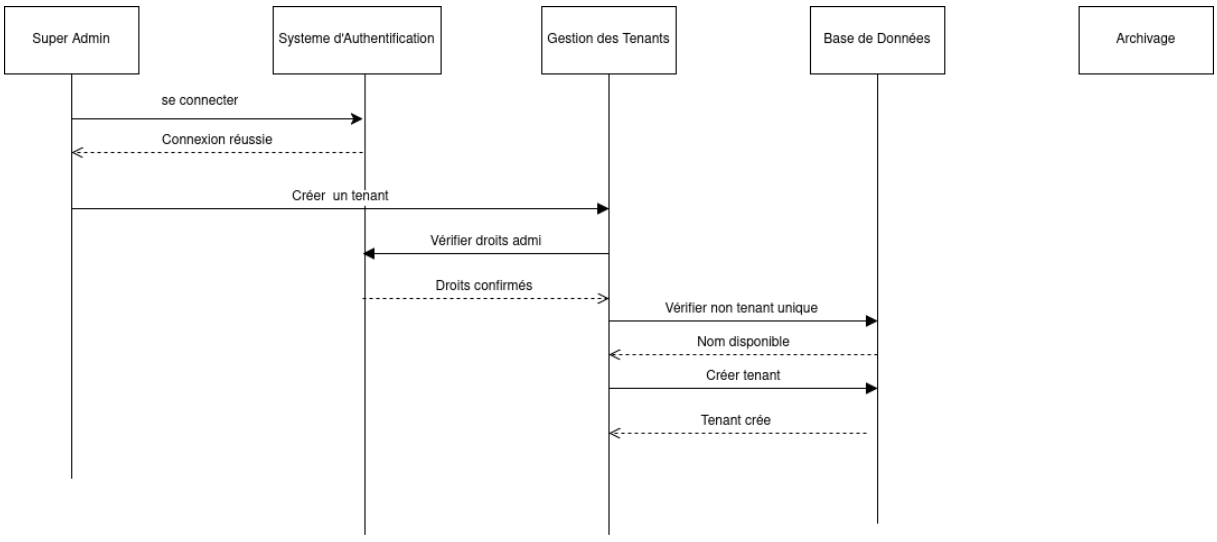

FIGURE 5 – Diagramme de sequences systeme : créer un tenant

3.2.5 Diagramme de sequences systeme : modifier les informations d'un tenant

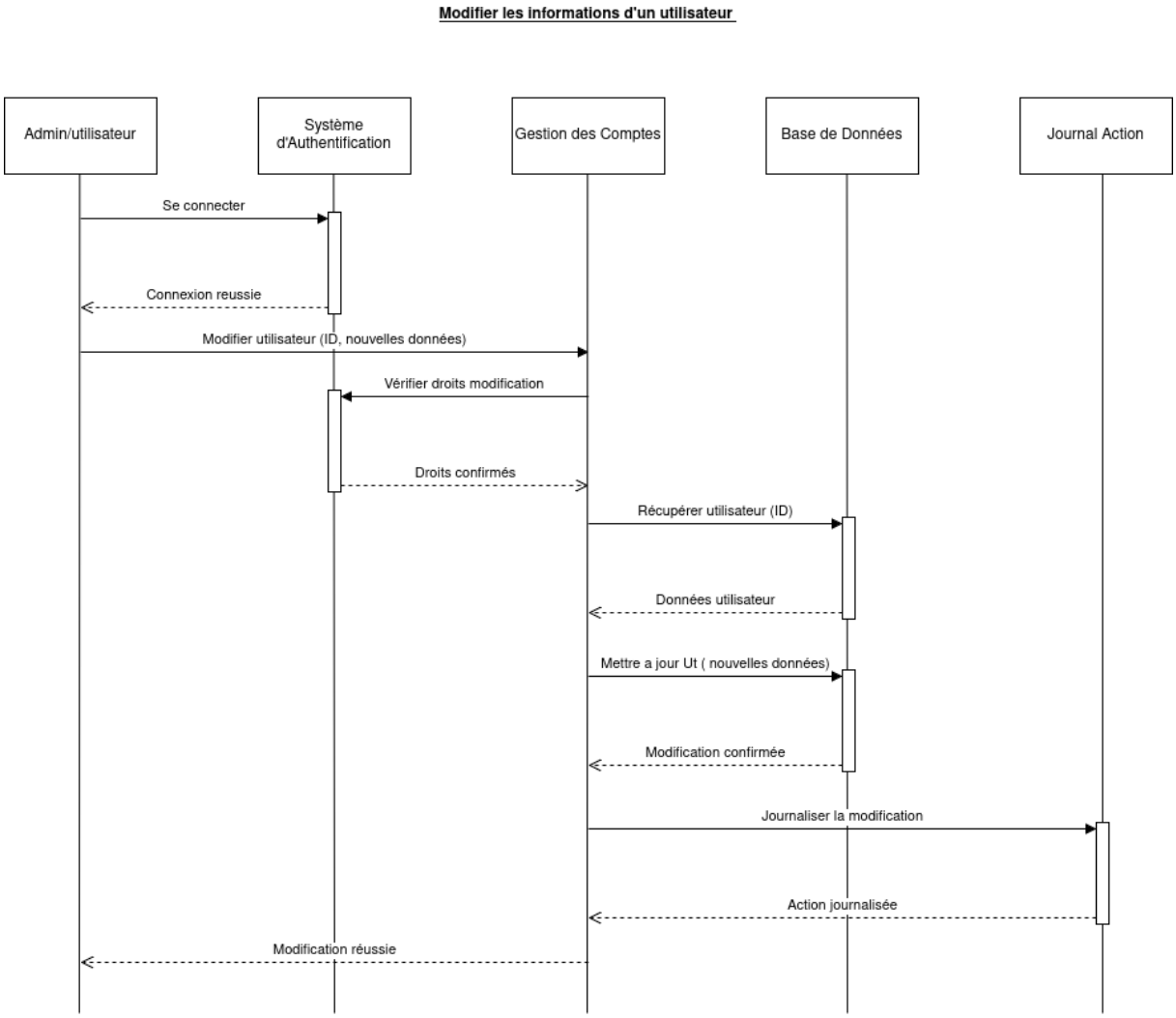

FIGURE 6 – Diagramme de sequences systeme : modifier les informations d'un tenant

3.2.6 Diagramme de sequences systeme : ajouter un utilisateur a un tenant

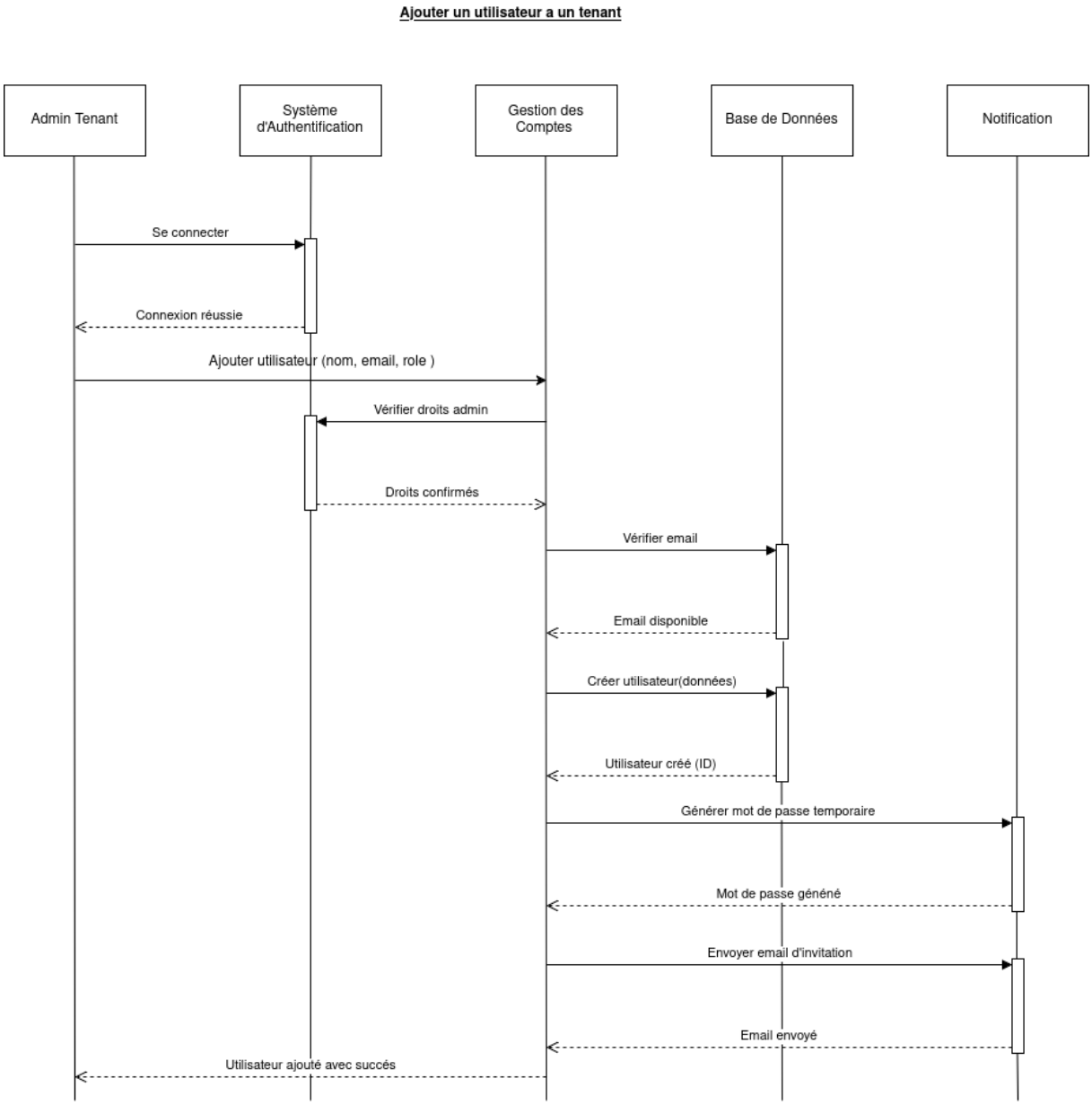

FIGURE 7 – Diagramme de sequences systeme : ajouter un utilisateur a un tenant

3.2.7 Diagramme de sequences systeme : creer un role

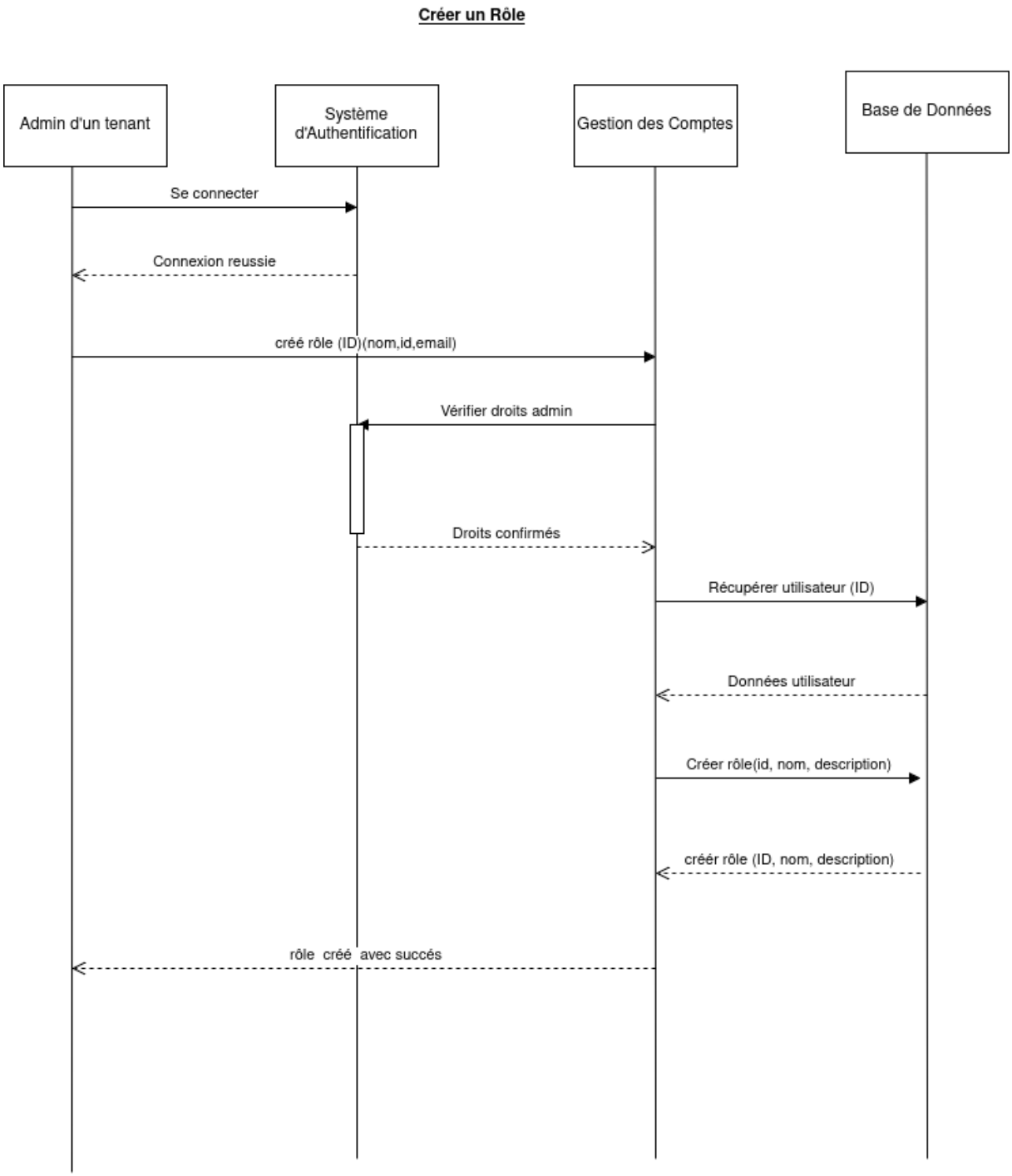

FIGURE 8 – Diagramme de sequences systeme : creer un role

3.2.8 Diagramme de sequences systeme : modifier un role

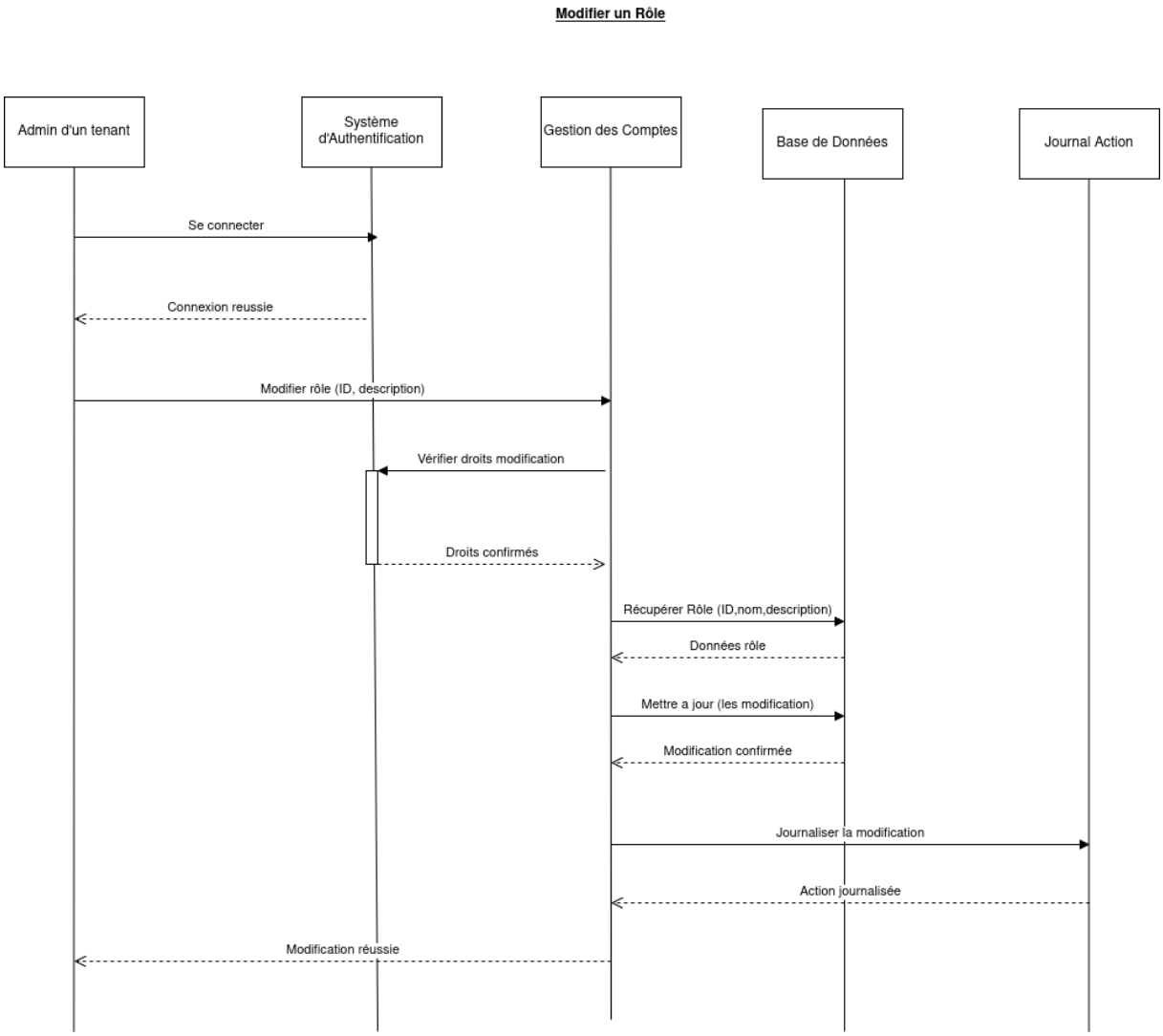

FIGURE 9 – Diagramme de sequences systeme : modifier un role

3.2.9 Diagramme de sequences systeme : consulter les information selon le role

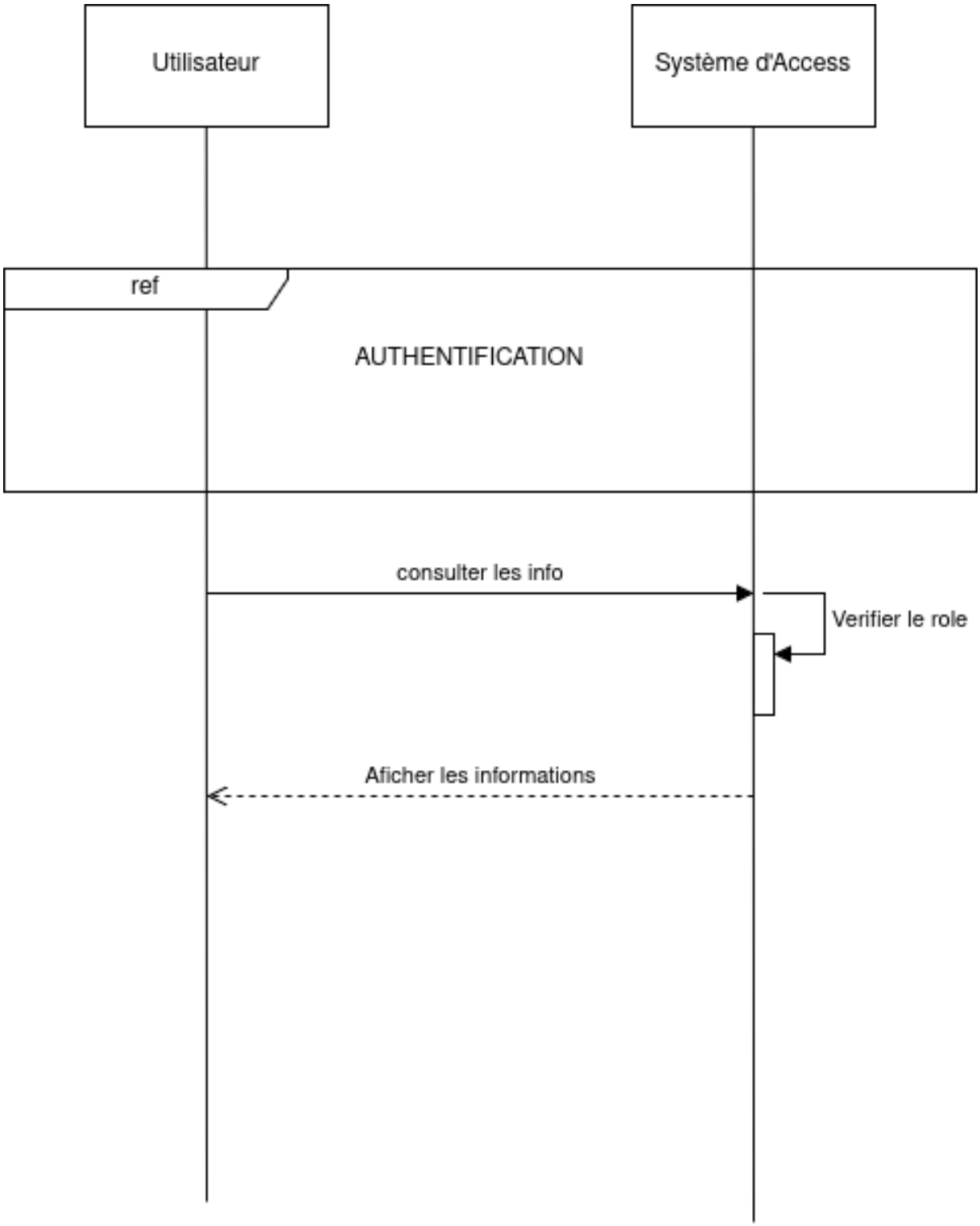

FIGURE 10 – Diagramme de sequences systeme : consulter les information selon le role

3.2.10 Diagramme de sequences systeme : supprimer un utilisateur

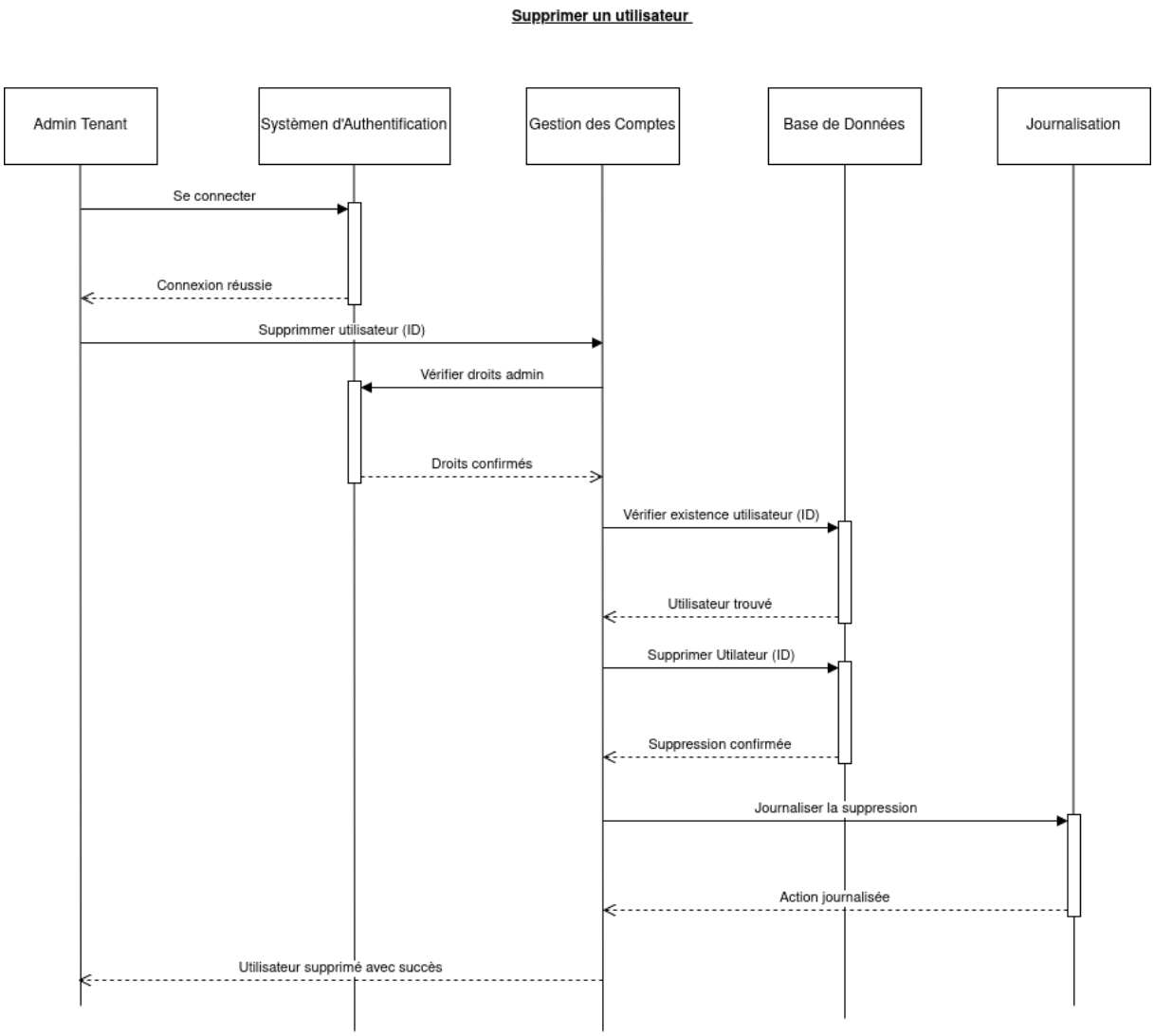

FIGURE 11 – Diagramme de sequences systeme : supprimer un utilisateur

3.3 Contraintes techniques

En s'appuyant sur la charte de développement et les spécifications techniques, les contraintes suivantes doivent être respectées :

- **Technologies obligatoires :** Java / Spring Boot pour le backend, PostgreSQL pour la base de données, Redis pour le cache optionnel, Next.js pour le frontend.
- **Architecture :** Conception inspirée LDAP-like avec support multi-tenant. Architecture évolutive vers microservices.
- **Déploiement et CI/CD :** Docker pour la containerisation, GitHub Actions pour intégration continue et déploiement automatique.
- **Performance :** Réponse des requêtes critiques (authentification, recherche utilisateurs) < 300 ms.
- **Sécurité :** Chiffrement des mots de passe (BCrypt ou Argon2), isolation stricte des tenants, contrôle d'accès basé sur les rôles.

— **Qualité du code :** Respect des conventions de nommage, SOLID, design patterns, tests unitaires avec coverage 80%, gestion des exceptions spécifiques.

3.4 Contraintes pédagogiques

— L'API doit illustrer les bonnes pratiques d'ingénierie logicielle et de conception multi-tenant.
— Le projet doit permettre aux étudiants de développer des compétences sur un stack moderne et intégré (Java/Spring Boot, PostgreSQL, Redis, Next.js, Docker, CI/CD).
— Documentation complète et code commenté conformément à la charte de développement.

3.5 Exigences non fonctionnelles

— **Sécurité et confidentialité** : données isolées par tenant, journalisation complète, pas de log des mots de passe.
— **Disponibilité** : disponibilité du système 99%.
— **Scalabilité** : architecture prête à évoluer vers un système distribué multi-service.
— **Maintenabilité** : code modulaire, testable et documenté.
— **Compatibilité** : API conforme aux standards REST, accessible via web et mobile.
— **Traçabilité** : toutes les actions sensibles doivent être enregistrées et consultables pour audit.

4 Tests et Validation

4.1 Objectif de la phase de test

La phase de tests et de validation vise à garantir que l'API multi-tenant développée répond aux exigences fonctionnelles, techniques et de sécurité définies dans ce cahier des charges. Elle permet de vérifier que :

— le système est fiable, sécurisé et conforme aux spécifications ;
— les différentes entités (tenants, utilisateurs, rôles, permissions) interagissent correctement ;
— chaque organisation (tenant) reste isolée des autres ;
— les rôles et permissions assurent un contrôle d'accès cohérent ;
— la performance et la stabilité du système sont acceptables.

4.2 Types de tests à réaliser

1. **Tests unitaires** Vérifient le bon fonctionnement de chaque composant isolément (services, contrôleurs, entités).
   — Exemple : vérification de la création d'un utilisateur dans le bon tenant.
   — Outil : `JUnit 5`.
2. **Tests d'intégration** Vérifient l'interaction entre plusieurs composants du système.
   — Exemple : enchaînement des étapes d'authentification → génération de token → accès à une ressource.
   — Outils : `Spring Boot Test`, `MockMvc`.
3. **Tests fonctionnels** Permettent de valider les cas d'utilisation métier à partir des besoins utilisateurs.
   — Exemple : un administrateur d'organisation peut créer un rôle et l'assigner à un utilisateur.
4. **Tests de sécurité** Garantissent la protection des données et le respect des règles d'accès.
   — Vérification de l'isolation des tenants.
   — Tests d'accès interdits (403 Forbidden).
   — Chiffrement et validation des mots de passe (BCrypt).
5. **Tests de performance et de charge** Évaluent la capacité du système à répondre rapidement sous forte sollicitation.
   — Mesure du temps de réponse moyen (< 300 ms pour l'authentification).
   — Simulation de plusieurs connexions simultanées.
   — Outils : `Postman`, `JMeter`.

4.3 Critères de validation

Le système sera considéré comme validé si :
— 100% des tests unitaires critiques sont réussis ;
— tous les cas d'utilisation fonctionnels sont couverts et validés ;
— aucun utilisateur ne peut accéder aux données d'un autre tenant ;
— les permissions sont appliquées strictement selon les rôles définis ;
— la documentation technique et fonctionnelle est complète et à jour.

4.4 Procédure de validation finale

— Revue de code collective et validation par le groupe projet.
— Exécution de la suite complète de tests automatisés.
— Démonstration pratique devant le jury : authentification, gestion des rôles, isolation multi-tenant.
— Signature du procès-verbal de validation technique.

4.5 Outils de test et suivi qualité

— **JUnit 5** : tests unitaires.
— **Spring Boot Test / MockMvc** : tests d'intégration.
— **Postman** : tests manuels et automatisés des endpoints.
— **JMeter** : tests de charge et de performance.
— **GitHub Actions** : intégration continue (CI/CD) avec exécution des tests à chaque commit.

5 Maintenance et Évolutions

5.1 Objectif de la maintenance

L'objectif de la maintenance est de garantir la continuité de service, la sécurité et la performance de l'API multi-tenant sur le long terme. Elle vise également à faciliter l'ajout de nouvelles fonctionnalités, la correction des anomalies et l'adaptation du système aux besoins futurs des organisations utilisatrices.

5.2 Types de maintenance prévus

- **Maintenance corrective** : Correction rapide des anomalies détectées après déploiement, notamment celles liées à la gestion des accès ou à l'isolation des tenants.
- **Maintenance préventive** : Surveillance continue du système à travers des outils de monitoring (ex. Prometheus, Grafana) afin d'anticiper les pannes et problèmes de performance.
- **Maintenance évolutive** : Ajout de nouvelles fonctionnalités ou ajustement des services existants sans remettre en cause l'architecture globale. Exemple : intégration d'un moteur de recherche `Elasticsearch`, ou ajout d'un service de notification via `Kafka`.
- **Maintenance adaptative** : Adaptation du système aux évolutions technologiques (versions de Java, Spring Boot, PostgreSQL, etc.) et aux contraintes des environnements d'exécution (cloud, conteneurs Docker, etc.).

5.3 Outils et pratiques de maintenance

La maintenance reposera sur un ensemble d'outils et de bonnes pratiques issues de la culture DevOps :

- **Gestion du code source** : GitHub (branches par fonctionnalité, pull requests, code review).
- **Intégration continue** : GitHub Actions pour l'exécution automatique des tests à chaque modification du code.
- **Surveillance et logs** : Utilisation de `Spring Boot Actuator` pour la supervision des endpoints, et d'un **Audit Log** pour tracer les actions (IP, timestamp, utilisateur, ressource).
- **Gestion des dépendances** : Maven/Gradle pour la cohérence des versions et la sécurité des packages.
- **Containerisation et déploiement** : Docker Compose pour les environnements de test et production.

5.4 Évolutivité et extensibilité du système

L'architecture du projet repose sur les microservices, ce qui permet :

- d'ajouter de nouveaux services (notification, reporting, analyse de données) sans impacter le cœur de l'API ;
- de déployer indépendamment chaque module pour une meilleure tolérance aux pannes ;
- d'adopter une approche "scalable" où chaque service peut être répliqué selon la charge.

5.5 Documentation et transfert de compétences

Une documentation technique complète sera maintenue sur un dépôt GitHub public ou privé :

- Documentation des endpoints API via `Swagger/OpenAPI`.
- Diagrammes UML (cas d'utilisation, classes, séquences) et schémas de la base de données.
- Guide de déploiement (Dockerfile, docker-compose.yml).
- Manuel utilisateur (interface d'administration et gestion des rôles).

5.6 Plan de continuité et perspectives d'évolution

À long terme, le projet pourra évoluer vers :

- une architecture distribuée complète (Spring Cloud, Kubernetes) ;
- l'intégration d'un annuaire LDAP réel pour la fédération des identités ;
- l'interfaçage avec une blockchain publique légère pour la traçabilité et la confiance numérique ;
- une interface web et mobile unifiée (PWA) utilisant les données locales via DuckDB et DuckDB-Wasm.

6 Conclusion

Le présent cahier des charges décrit la conception et la mise en œuvre d'une API multi-tenant inspirée du protocole LDAP, destinée à la gestion des utilisateurs, des rôles et des permissions dans un environnement distribué et sécurisé. Ce projet s'inscrit dans une approche pédagogique et technologique visant à renforcer les compétences des étudiants dans le développement d'applications modernes, résilientes et orientées services.

Le système proposé permettra à plusieurs organisations (tenants) de coexister sur une même plateforme, tout en garantissant une stricte isolation des données et un contrôle d'accès basé sur les rôles et les permissions. L'architecture repose sur les principes des microservices, l'intégration continue, le monitoring et les bonnes pratiques de développement durable du code. Ces choix techniques assurent non seulement la fiabilité du système, mais aussi sa capacité à évoluer dans le temps sans dégrader la qualité du service.

D'un point de vue fonctionnel, cette API se veut un socle central pour la gestion des identités numériques et la gouvernance des accès au sein de différentes structures. Elle répond à un besoin croissant de centralisation, de sécurité et de confiance numérique, dans un contexte où les organisations sont de plus en plus interconnectées et soumises à des exigences élevées de confidentialité.

Sur le plan pédagogique, le projet favorise l'apprentissage collaboratif et la maîtrise de technologies industrielles telles que Spring Boot, PostgreSQL, Docker, GitHub Actions, ainsi que des outils de monitoring et de performance. Il constitue une opportunité concrète pour les étudiants d'appliquer les principes du DevOps, de l'ingénierie logicielle et de la conception distribuée dans un cadre professionnel simulé.

Enfin, ce projet se distingue par sa vision durable et évolutive. Il ouvre la voie à des intégrations futures telles que :

- l'interconnexion avec un annuaire LDAP ou une blockchain légère pour la traçabilité ;
- l'exploitation locale des données via DuckDB pour des applications résilientes et low-latency ;
- le déploiement d'interfaces web et mobiles progressives (PWA) adaptées aux besoins de chaque tenant.

En conclusion, ce projet illustre une démarche complète d'ingénierie logicielle moderne : de la conception à la validation, en passant par la sécurité, la scalabilité et la maintenance continue. Il constitue une base solide pour des applications futures à fort impact social et technologique.