Projet API

Réalisé par :

- **Julien Zenner**

Enseignant : Olivier PERRIN

Université de Lorraine – Institut des Sciences du Digital Management & Cognition – M2 MIAGE SID

Bâtiment PHS – 13 rue Michel Ney, 54000 NANCY

Sommaire

Table des matières

Réalisation

Outils utilisés

Consul

Consul est utilisé pour la découverte des microservices Spring lancés. Ainsi vu que le projet utilise openfeign ce service discovery est très utile puisque grâce à Consul feign sait sur quel port et avec quel hostname et/ou adresse ip les microservices sont lancés.

Consul est démarré à l'aide de Docker. Aucune configuration spécifique dans Consul est nécessaire.

RabbitMQ

RabbitMQ est un bus de messages qui implémente le protocole Advanced Message Queuing (AMQP). Sa fonction est de faire communiquer entre eux des programmes différents, potentiellement écrits dans différents langages.

Le système de Message RabbitMQ a été utilisé dans ce projet pour effectuer des appels asynchrones et ainsi améliorer la tolérance aux pannes de ce projet. En effet, je l'utilise sur toutes les méthodes POST, PUT, PATCH, DELETE pour ne pas perdre l'informations au cas où le/les microservice cible devant consommer le/les messages ne sont pas disponibles au moment de cette requête http REST.

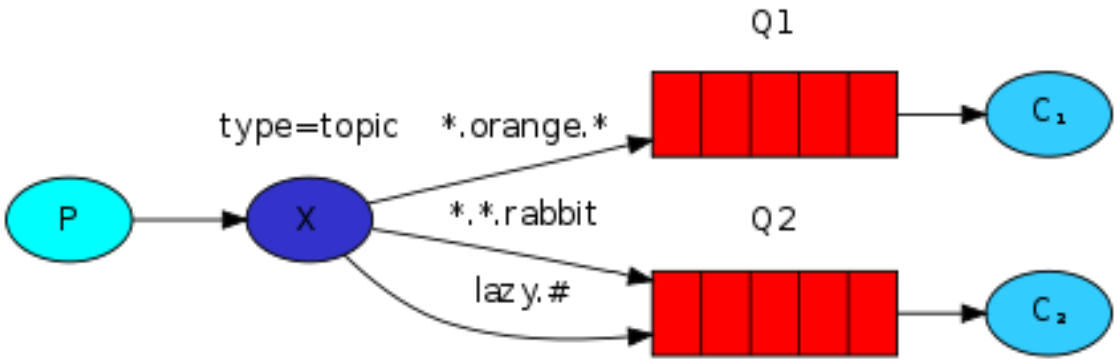


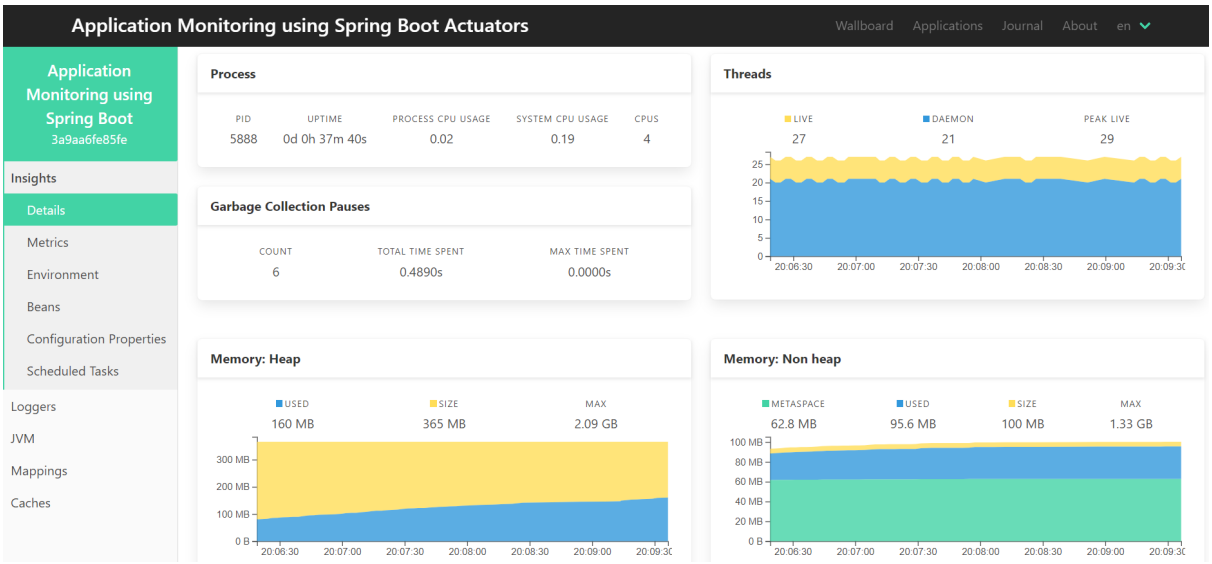
Figure 1 : Exemple d'architecture RabbitMq

Dans ce projet, RabbitMQ a été configuré de cette sorte : il y a un exchange de type topic. Chaque message arrive dans cet exchange avec une routing key. Cette routing key permet d'envoyer ce message dans une queue spécifique. Sur ces queues un ou plusieurs consommateurs peuvent être connecté. Si un message arrive dans une queue un de ces consommateurs va s'activer pour effectuer le traitement.

SpringBootAdmin

Spring Boot Admin est une application Web utilisée pour gérer et surveiller les applications Spring Boot. Chaque application est considérée comme un client et s'enregistre auprès du serveur d'administration. SpringBootAdmin utilise le service discovery consul pour découvrir tous les microservices à superviser. Pour afficher les données métriques SpringBootAdmin utilise la dépendance spring boot actuator des microservices à superviser pour afficher les données.

Chaque microservice de ce projet contient la dépendance spring boot actuator pour assurer la supervision. Tous les endpoints ajouté par spring boot actuator sont utilisables. En production, il faudra revoir cette partie afin de ne pas exposer des données trop sensibles. Nous pouvons avoir des informations sur la JVM, l'utilisation CPU, RAM, le garbage collector, etc.



PostgreSQL

PostgreSQL est mon système de base de données. Chaque microservice possède sa propre base postgres. Ainsi, les microservices sont faiblement couplés. Un script sql est joué pour populariser la base de données à chaque lancement d'un microservice. Ayant plusieurs microservices, les bases de données sont lancés automatiquement à l'aide d'un docker-compose fournis dans le livrable. PostgreSQL a été préféré à une base de données H2 puisque la base H2 est lancée en même temps que le microservice embarqué dans ce dernier. La base de données H2 peut poser des problèmes lors de la duplication d'un même microservice qui possèdera alors sa propre base de données dans chaque microservice dupliqué et donc à ce moment, il peut avoir un problème au niveau des données.

Docker

Docker est utilisé dans l'ensemble de ce projet. En effet, il permet dans un premier temps de lancer tous les services nécessaires au bon fonctionnement des microservices spring. (consul, base de données, rabbitmq, …), un docker-compose est disponible à la racine du dossier. Dans un second temps dans l'arborescence des projets microservices se trouve un fichier Dockerfile permettant de conteneuriser les microservices. Ceci permet un avantage non-négligeable puisqu'il permet de lancer les microservices dans un environnement isolé indépendant du système d'exploitation de l'hôte, un autre fichier docker-compose est disponible afin de démarrer tous les microservices dans docker.

Découpage micro-services

Le découpage en microservice dans ce projet a été effectué selon un découpage métier. Ainsi, nous avons deux microservice métier différents utilisateurs et cours.
Le microservice utilisateur est en charge de la gestion utilisateur. Le microservice cours est en charge de la gestion des cours et épisodes de ces cours présent dans l'appli.
A cela s'ajoute une gateway zuul qui est en quelque sorte l'entrypoint de l'api permettant de rediriger les requêtes vers le microservice cible ou d'envoyer le message à rabbitmq.
Un microservice OAuth qui est en charge de la gestion des comptes de connexion utilisateurs. Grâce à ce microservice les utilisateurs bénéficient d'un jeton d'accès JWT permettant aux utilisateurs de naviguer dans l'api selon le droit qu'ils ont dans cette api. Chaque microservice est sécurisé à l'aide de spring-security. Cette dépendance fonctionne de pair avec le jeton d'accès.
La navigation entre les endpoints est facilitée à l'aide de l'utilisation de Hateoas.

Service Utilisateur

Utilisateur-service est le micro-service de gestion des utilisateurs. Il est connecté à la base de données PostgreSQL utilisateur. Il permet de créer, modifier, supprimer, récupérer un ou plusieurs utilisateurs. Un validator et des annotations de validation sont présentes sur les classes Input, pour que les données restent cohérentes en base de données et également éviter les erreurs de sauvegarde en base de données.

Il a également des routes permettant de récupérer pour un utilisateur donné, les cours pour lesquels il s'est abonné et les épisodes visionnés par cet utilisateur. Un client Feign a été mis en place pour communiquer directement avec le microservice cours pour récupérer les cours d'un utilisateur. Ces méthodes sont accessibles, par la méthode GET, si l'utilisateur connecté est l'utilisateur en question, pour un admin celui-ci a le droit d'effectuer cette requête pour tout le monde.

Il est également relié à OAuth via le RabbitMQ. Lors de la création d'un utilisateur le microservice envoie un message à OAuth pour créer le profil de cet utilisateur. Ce message a pour but que cet utilisateur puisse se connecter à l'api.

Une requête POST est aussi présente pour qu'un utilisateur puisse s'enregistrer sur un cours. Il est nécessaire de passer plusieurs paramètres : l'id de l'utilisateur, l'id du cours et une carte bancaire si le cours est payant. La première étape est donc la récupération du cours en question grâce à Feign. Si le cours est payant et qu'il n'y a pas de carte bancaire, ou que la carte bancaire finit par un chiffre impair alors la réponse sera de type BadRequest, sinon le cours sera ajouté à l'utilisateur.
Ce microservice est également consommateur de plusieurs queues de RabbitMQ, c'est-à-dire que si un message est envoyé dans la queue, il le consomme et exécute le code associé. Ces queues en question sont pour la création, modification totale ou partielle, suppression d'un utilisateur ou encore l'enregistrement d'un utilisateur sur un cours.
Chaque queue a un nom unique et possède les mêmes méthodes adaptées que pour l'API REST, sauf pour l'enregistrement du cours, il manque à ce moment-là, la récupération de l'objet cours au niveau de la méthode asynchrone.
La suppression d'un utilisateur est une suppression logique, c'est-à-dire que le champs statut passe d'Actif à Supprimé.

Service Cours

Cours-service est le microservice de gestion des cours et épisodes que contient un cours. Le service permet de créer, modifier, supprimer, consulter les cours et les épisodes associés.
Cours service permet aussi de créer, modifier, supprimer, consulter les épisodes.
Au niveau de la base de données, il n'y a pas de relation explicite entre ces deux objets afin que plus tard, si le besoin est présent pouvoir séparer ces deux métiers en deux microservices différents.

Pour pouvoir créer un épisode, il est essentiel de passer en argument l'id du cours. Deux méthodes après sont présentes une pour un épisode avec un lien url et une autre méthode avec un fichier vidéo cette méthode elle, requiert l'utilisation d'un MultiPart.
Lorsqu'un cours est supprimé, tous les épisodes associés sont eux aussi supprimés. La méthode GET sur un épisode d'un cours envoie un évènement à rabbitMQ afin d'être consommer par le microservice utilisateur afin d'enregistrer cet épisode comme vu par l'utilisateur. Une partie asynchrone est présente dans ce microservice afin de gérer toute la partie création, modification, suppression d'un cours ou épisode.

Service OAuth

OAuth service est le microservice de gestion de la connexion des utilisateurs. L'utilisateur va devoir renseigner sont long mot de passe et le microservice retournera un jeton d'accès et un jeton de renouvellement du jeton d'accès. Ce microservice permet à l'aide de ce jeton d'accès d'intégrer des notions de sécurité dans les autres microservices. Ainsi, un utilisateur ADMIN n'aura pas les même droits qu'un utilisateur lambda.

Ce microservice est simple il permet de créer un compte de connexion pour un utilisateur. Il est de ce fait consommateur de la queue 'new-user' ce qui permet que lorsqu'un utilisateur créé son compte ce dernier créer aussi un compte de connexion lié.

Actuellement, il existe seulement deux rôles dans l'application : ROLE_USER et ROLE_ADMIN, mais cela peut être modifié pour une mise en production.

Service Gateway

Gateway service est le point d'entrée de l'application. En effet les routes REST des microservices cours et utilisateurs sont présentes dans ce service. C'est l'entrypoint, il ne comporte donc aucun code métier, il sert juste de passe-plat.

Ce service fonctionne avec les outils Feign, Hystrix, RabbitMQ et Consul. Ainsi en utilisant ce point d'entré le load-balancing, failover sont pris en charge. Feign se charge de communiquer avec consul pour savoir où se trouve les microservices pour rediriger la requête HTTP, le load-balancing entre les microservices est aussi géré par feign. Hystrix est notre coupe-circuit au cas où un des microservices n'est pas disponible pour éviter l'échec (erreur 500).

Gateway service est aussi le producteur de message pour le rabbitMQ. Ainsi, plus besoin de se soucier si les microservices métiers sont disponible, car les requêtes http POST, PUT, PATH, DELETE sont effectués en asynchrone à partir de la gateway. L'utilisation de l'asynchrone est bénéfique puisqu'il permet de ne pas perdre les requêtes au cas où les microservices métiers consommateurs ne sont pas disponibles.

Amélioration

Réception message RabbitMQ

L'amélioration qui est possible d'apporter au système de messages actuellement présent est le fait de pouvoir récupérer des objets d'un autre microservice sans casser le principe asynchrone, avantage du système de message.

Par exemple, dans le microservice utilisateur, lors de l'enregistrement d'un utilisateur sur un cours, en passant par le système de message, on perd les règles métiers correspondant à la carte bancaire et au prix du cours, car il est impossible de récupérer de façon asynchrone le cours or, j'ai privilégié l'asynchrone à la place des règles métiers qui l'aurait cassé.

Hateoas

Une amélioration d'hateoas est la génération des liens, en effet en passant par la Gateway les liens générés renvoient vers l'adresse du microservice en question et non l'adresse de la gateway. Cela peut poser divers problèmes si par exemple les microservices sont totalement sécurisés avec aucune entrée possible en dehors de la gateway (par exemple sur une mise en pratique sur le cloud).

Oauth

Une amélioration est possible à ce niveau en remplaçant le microservice oauth par un outil dédié, plus robuste tel que Keycloak.

Base de données

L'amélioration possible pour les bases de données serait de clustériser chaque base de données utilisée par un microservice : utilisateur, cours/épisode, oauth.

La clustérisassions des bases de données n'est pas quelque chose de simple, mais pourra améliorer la haute disponibilité de ce projet et ainsi éviter un single point of failure évident.

En effet, actuellement, si une base de données vient à s'arrêter inopinément le microservice relié à cette base de données devient inutilisable.

Conclusion

J'ai trouvé ce projet très intéressant, car mettre en place une architecture microservice depuis le début jusqu'à la finalité avec un exemple concret est très enrichissant.

De par mon alternance, j'ai pu déjà m'exercer aux architectures microservice avec spring. Cependant, j'ai vu divers aspects que je ne connaissais pas ou maitrisais mal comme les appels asynchrones via rabbitMQ qui pour l'instant en entreprise servaient simplement à envoyer des mails, génération de documents, etc. Hystrix et Hateoas ne sont pas mise en place dans les architectures que j'ai pu développer dans l'équipe où je suis à Inetum. Mais je pense qu'une évolution pour intégrer ces technologies est à prévoir.

Je n'ai pas forcément eu de difficulté à réaliser ce projet, puisque je connaissais déjà bien les architectures microservice, mais il y a eu quelques complications avec rabbitmq, hystrix et hateoas, c'est ce que j'ai trouvé le plus intéressant à devoir résoudre.

Annexes

Schéma architecture

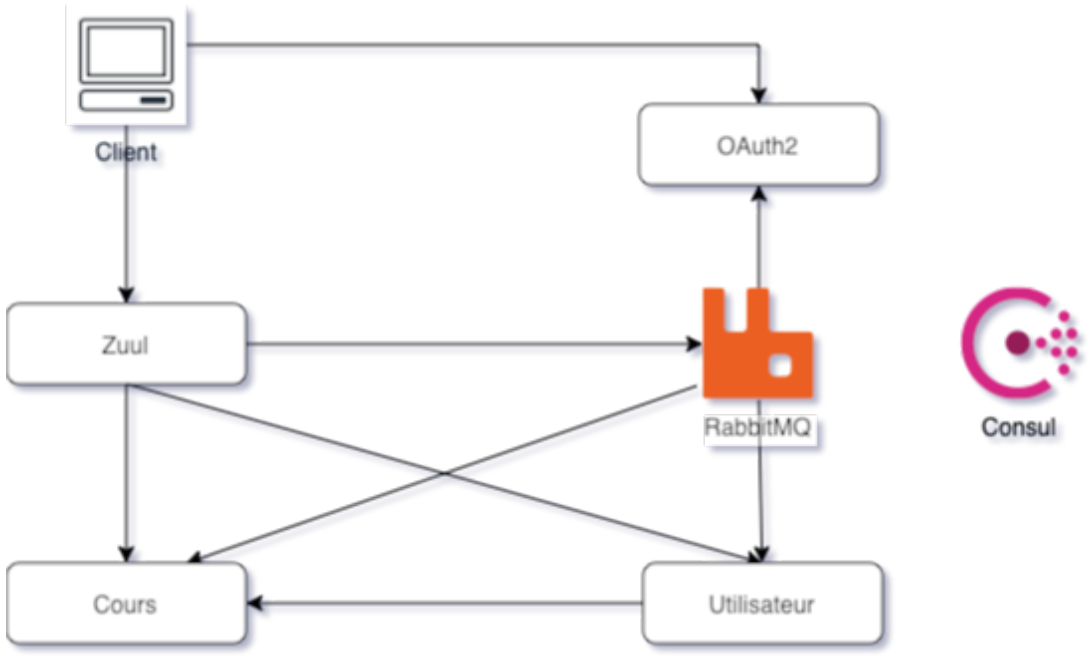


Diagramme de séquence : enregistrement d'un cours pour un utilisateur

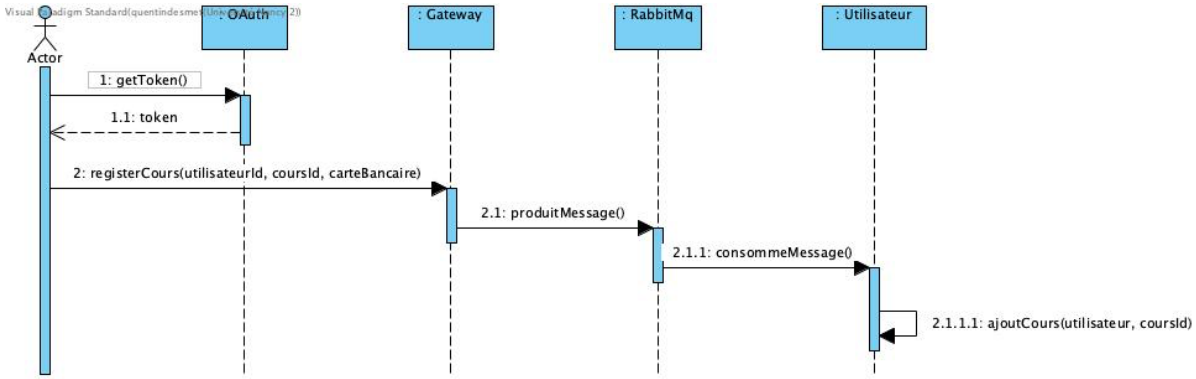


Diagramme de séquence : création d'un utilisateur

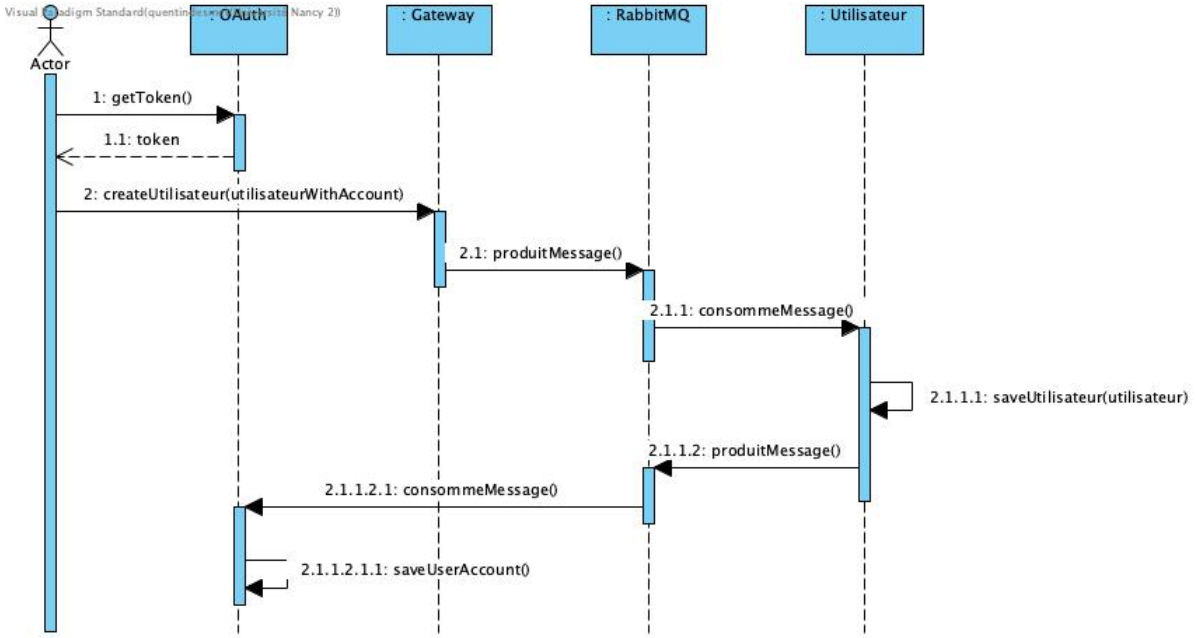


Diagramme de séquence : récupération utilisateur avec ses cours et épisodes

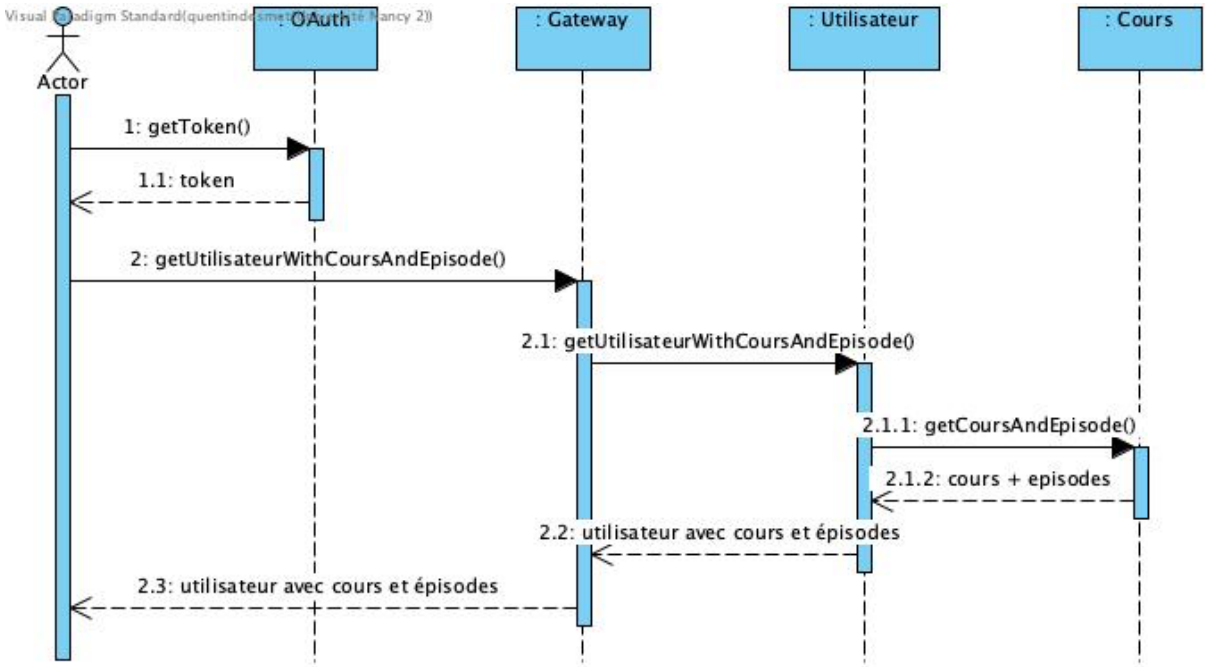


Diagramme de séquence : récupération d'un épisode qui effectue la vision de ce épisode par l'utilisateur connecté

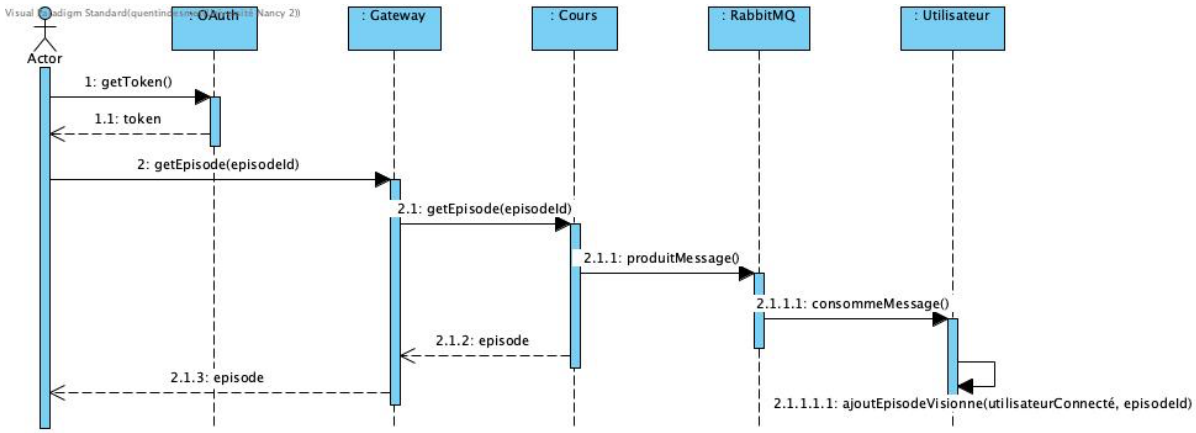


Diagramme de séquence : suppression d'un cours