Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Chancellerie fédérale ChF

Secteur Transformation numérique et gouvernance de l'informatique TNI

API : L’idempotence

Principe d’architecture 10 (AP10) de l’architecture API de la Confédération

Classification	Non classifié
Statut	En cours de revue
Nom du programme	Stratégie Administration fédérale numérique Priorité 1 : Penser et concevoir l’administration numérique en réseau
Responsable de l’initiative	Karen Dijkstra, BK
Version	1.0
Date	9 octobre 2024
Mandant	Transformation numérique et gouvernance de l’informatique (TNI)
Autrice	Karen Dijkstra
Groupes spécialisés	Groupe Architecture API de la Confédération, Conseil de l’architecture de la Confédération (ABB)
Approuvé par	Andreas Spichiger (BK)

Management summary

Ce document analyse en profondeur le principe d'architecture 10 (AP10) de l'architecture API de la Confédération, centré sur l'idempotence dans les API. Ce principe est essentiel pour garantir la robustesse et la fiabilité des systèmes distribués.

L'idempotence est un concept issu des mathématiques, qui permet à une opération de produire le même effet indépendamment du nombre de fois qu'elle est exécutée. Dans le contexte des API RESTful, elle se traduit par des méthodes HTTP spécifiques comme GET, PUT et DELETE, qui assurent que les requêtes répétées n'altèrent pas l'état du système au-delà de la première requête. Le texte examine également les méthodes non-idempotentes telles que POST et PATCH, et propose des solutions pour gérer leur impact, notamment l'utilisation de jetons idempotents.

Les avantages de l'idempotence incluent une meilleure gestion des erreurs, une reprise simplifiée après des défaillances et une conception de système plus robuste. Cependant, elle pose aussi des défis en termes de gestion de l'état, de sécurité et de performance. Le document fournit des directives pratiques pour surmonter ces défis, comme l'utilisation de jetons idempotents et des stratégies de contrôle de version.

Ce rapport est destiné aux développeurs, architectes et décideurs techniques, offrant des lignes directrices pratiques et des perspectives de recherche pour implémenter efficacement l'idempotence dans les API. En détaillant le principe d'architecture 10 (AP10), il soutient les initiatives de transformation numérique et de gouvernance de l'informatique au sein de l'Administration fédérale, en ligne avec la Stratégie Administration fédérale numérique.

L'étude fournie ici vous guidera dans la compréhension et la mise en œuvre de l'idempotence, améliorant ainsi la fiabilité et la robustesse de vos systèmes API. Nous vous invitons à explorer ce texte pour découvrir comment appliquer ces principes à vos projets.

Table des matières

1 Introduction

1.1 Définition de l'idempotence

Selon le « Quasar Enterprise[1] » *les opérations idempotentes se caractérisent par le fait qu'un appel répété avec les mêmes arguments a le même effet qu'un appel unique. Exemple : il suffit d'annuler une fois ; un deuxième appel à l'opération d'annulation reste sans effet et ne cause donc aucun dommage. L'idempotence a un effet positif sur la robustesse des applications, car elle permet d'éviter que des appels multiples d'opérations, effectués par erreur, n'entraînent des problèmes. L'orchestration de processus robustes à partir d'opérations est plus simple lorsque celles-ci sont idempotentes.*

Par ailleurs, l'idempotence est un concept qui trouve ses racines dans les mathématiques et qui a été appliqué avec succès dans le développement d'API. Dans ce contexte, l'idempotence se réfère à la propriété selon laquelle une requête HTTP spécifique peut être envoyée plusieurs fois sans changer l'état du système au-delà de la première requête. Les méthodes HTTP comme GET, PUT, et DELETE sont conçues pour être idempotentes.

Considérez une méthode DELETE pour supprimer un utilisateur spécifique dans une base de données. Si cette opération est idempotente, l'utilisateur sera supprimé lors de la première requête, et les requêtes suivantes auront le même effet (l'utilisateur restera supprimé).

En complément de cet exemple, il est également important que le serveur renvoie une réponse cohérente lors des requêtes répétées. Si la première requête DELETE a réussi, mais que la réponse n'a jamais atteint le client, le serveur doit, lors d'une requête répétée, renvoyer un code 2xx (OK) indiquant que l'opération a bien été effectuée, plutôt qu'un code 404 (Not Found) signalant que l'utilisateur n'existe plus. Cela garantit que l'API se comporte de manière prévisible et fiable dans les environnements où la transmission réseau est incertaine.

De plus, si la spécification de l'API prévoit que la réponse à une requête DELETE doit renvoyer l'objet supprimé, le serveur doit être capable de renvoyer cet objet même lors d'une requête répétée. Cela garantit que le client peut obtenir une réponse complète et conforme, quelle que soit la fiabilité de la transmission initiale.

En somme, l'idempotence permet d'assurer la fiabilité des systèmes distribués. Elle permet aux clients de répéter en toute sécurité les requêtes sans effet secondaire, ce qui est particulièrement important dans les environnements où les réseaux peuvent être peu fiables.

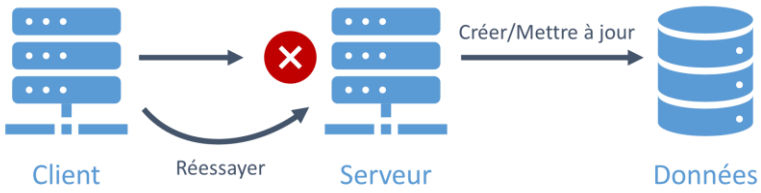

Figure 1 : Principe de fonctionnement de l'idempotence

Du point de vue du client, il est difficile de distinguer si une requête ou sa réponse a été perdue, car aucune inspection directe du serveur n'est généralement possible. Cela complique également la détection pour tout composant réseau impliqué dans la transmission. Dans ce contexte, un schéma de transmission de type « at least once » est souvent utilisé.

[1] Quasar Enterprise – Anwendungslandschaften serviceorientiert gestalten; Engels, Hess, Humm, Juwig, Lohmann, Richter, Voss, Willkomm; sd&m; dpunkt,verlag; ISBN 978-3-89864-506-5

Dans le cadre de la numérisation, qui implique des processus automatisés, l'objectif est de minimiser les interactions manuelles. Cela nécessite également une gestion des erreurs qui soit largement automatisée. Ce type de gestion peut être considérablement simplifié en mettant en place des mécanismes de reprise automatique, évitant ainsi le traitement exhaustif et coûteux de tous les cas d'erreur possibles.

Selon la RFC 7231, une méthode de requête est considérée comme idempotente si l'effet prévu sur le serveur de plusieurs requêtes identiques avec cette méthode est le même que celui d'une seule requête de ce type. Parmi les méthodes de requête définies par la présente spécification, les requêtes PUT, DELETE et « sûres » sont idempotentes. Précisons que les requêtes sont considérées comme « sûres » si leur sémantique définie est essentiellement en lecture seule, c'est-à-dire que le client ne demande pas et n'attend pas de changement d'état sur le serveur d'origine à la suite de l'application d'une méthode sûre à une ressource cible. De même, l'utilisation raisonnable d'une méthode sûre ne devrait pas causer de préjudice, de perte de propriété ou de charge inhabituelle pour le serveur d'origine. La propriété idempotente ne s'applique qu'à ce qui a été demandé par l'utilisateur ; un serveur est libre d'enregistrer chaque demande séparément, de conserver un historique de contrôle des révisions ou d'implémenter d'autres effets secondaires non idempotents pour chaque demande idempotente. Ces méthodes se distinguent par le fait que la demande peut être répétée automatiquement en cas d'échec de la communication avant que le client ne puisse lire la réponse du serveur.

1.2 Objectif de l'étude

L'objectif de cette étude est d'explorer le concept d'idempotence faisant l'objet du principe d'architecture 10 (AP10) de l'architecture API de la Confédération[2], de comprendre ses implications pratiques, et de fournir des lignes directrices pour sa mise en œuvre efficace. L'étude vise à soutenir les développeurs, les architectes et les décideurs.

1.3 Portée de l'étude

La portée de cette étude inclut les API RESTful, largement utilisées dans l'industrie, ainsi que d'autres paradigmes d'API où l'idempotence peut être appliquée. Elle couvre les aspects théoriques, tels que les principes de base de l'idempotence, et les aspects pratiques, y compris les méthodes de mise en œuvre et des cas d'utilisation concrets.

2 Contexte et fondements théoriques

2.1 Historique de l'idempotence

L'idempotence est un concept qui a été étudié depuis longtemps dans divers domaines des mathématiques et de l'informatique. Son origine remonte à la théorie des algèbres, où elle a été utilisée pour décrire certaines propriétés des matrices et des opérations.

Dans le contexte des matrices, une matrice idempotente est une matrice qui, lorsqu'elle est multipliée par elle-même, donne le même résultat. Formellement, une matrice A est idempotente si $A^2 = A$. Cette propriété a des applications intéressantes dans la théorie des graphes, l'algèbre linéaire, et même dans l'étude des équations différentielles.

Dans les années 1970 et 1980, l'idempotence a trouvé une nouvelle application dans les systèmes distribués. À une époque où les réseaux étaient souvent peu fiables et sujets à des défaillances, l'idempotence est devenue un moyen d'assurer la fiabilité des opérations. Elle permettait aux systèmes de répéter des opérations sans crainte, contribuant ainsi à la résilience des systèmes.

[2] https://portal.collab.admin.ch/sites/104-BK-PuS/SP/Freigegebene%20Dokumente/Dokumentation/Aufgabenbereiche/Strategien/Strategieumsetzung%20ab%202020/SI-3_Once-Only_Prinzip/API-Architektur_Bund/Ergebnisse/API_Architektur_Bund/API-Architektur-Bund_V1.0.docx?d=w0ec916a17be94e0086e3820973823f44

Aujourd'hui, l'idempotence est un principe permettant de garantir que les requêtes peuvent être répétées sans effet secondaire, même en présence de pannes réseau ou de défaillances de serveur. Elle est ainsi devenue un pilier de la conception des systèmes distribués.

2.2 Principes de base

En mathématiques, une opération f est dite idempotente si l'application répétée de l'opération ne change pas le résultat, c'est-à-dire $f(f(x)) = f(x)$ pour tout x. Cette propriété a des implications profondes dans la théorie des ensembles, la logique, et l'algèbre.

Considérez la multiplication par zéro. Pour tout nombre x, $0 \cdot x = 0$, et $0 \cdot (0 \cdot x) = 0$ Ainsi, la multiplication par zéro est une opération idempotente.

Certaines opérations idempotentes peuvent également être commutatives et associatives, bien que ce ne soit pas une exigence. L'idempotence joue un rôle dans la définition de certaines structures algébriques, comme les semi-groupes idempotents. Par exemple, la fonction logique ET (∧) est idempotente car $x \wedge x = x$ pour tout x. Cette propriété est fondamentale dans la conception de circuits logiques et dans la simplification des expressions booléennes.

AND		
A	B	S
0	0	0
1	1	1

OR		
A	B	S
0	0	0
1	1	1

XOR		
A	B	S
0	0	0
1	1	0

XNOR		
A	B	S
0	0	1
1	1	1

L'idempotence dans les API REST est inspirée de ces principes mathématiques. Certaines méthodes HTTP, comme GET, PUT, et DELETE, sont conçues pour être idempotentes. Cette propriété est essentielle dans les environnements distribués où les défaillances réseau peuvent entraîner des requêtes dupliquées. Elle est également utilisée dans les systèmes de gestion de bases de données, où elle permet de garantir la cohérence des données lors de pannes. Les transactions idempotentes peuvent être réexécutées sans risque de corruption des données.

3 Méthodes idempotentes dans les API REST

Avant d'explorer l'idempotence dans le contexte des API REST, il est important de souligner que ce principe est d'ordre général et n'est pas limité aux seules API RESTful. L'idempotence est une propriété fondamentale qui s'applique à de nombreux types de systèmes et protocoles dans l'informatique distribuée. Elle est ici illustrée à travers les méthodes HTTP pour les API REST, mais son utilisation s'étend à d'autres paradigmes d'API et de communication, où la répétition de certaines opérations ne doit pas modifier l'état au-delà du premier appel.

Selon la RFC 7231, les méthodes HTTP idempotentes telles que GET, PUT et DELETE peuvent être appelées plusieurs fois sans effet secondaire. Par exemple, une requête GET peut être répétée sans changer le résultat, tandis qu'une requête POST peut avoir un effet différent à chaque appel. Le tableau ci-dessous propose un résumé des méthode HTTP qui seront développées dans les chapitres suivants.

Méthode HTTP	Idempotence	Description
GET	Oui	Récupère une ressource sans la modifier.
HEAD	Oui	Comme GET, mais sans le corps de la réponse.
OPTIONS	Oui	Retourne les méthodes HTTP supportées par le serveur.
TRACE	Oui	Retourne une boucle de diagnostic.
PUT	Oui	Met à jour une ressource ou la crée si elle n'existe pas.

Méthode HTTP	Idempotence	Description
DELETE	Oui	Supprime une ressource. Peut être appelée plusieurs fois avec le même résultat.
POST	Non	Crée une nouvelle ressource, démarre un processus ou met à jour une ressource.
PATCH	Non	Met à jour sélectivement une ressource.

3.1 Méthode HTTP idempotentes

Cette section explore les méthodes HTTP, leurs idempotences, leurs caractéristiques, et leurs implications dans la conception d'API.

GET : Requête de lecture	
Définition	La méthode GET est utilisée pour récupérer des données d'une ressource spécifique.
Idempotence	Elle doit toujours être idempotente, ce qui signifie que répéter une requête GET ne doit pas modifier l'état du système.
Utilisation	Parfaitement adaptée pour récupérer des données, elle est souvent utilisée avec des paramètres de requête pour filtrer les résultats.
Sécurité	Bien qu'idempotente, la méthode GET doit être utilisée avec prudence, en particulier pour les données sensibles, car les paramètres de requête peuvent être exposés dans les journaux du serveur.

Récupérer les détails d'un produit par son identifiant

- Idempotence : La méthode GET garantit que la récupération peut être répétée sans risque.
- Considérations : La pagination et le filtrage peuvent être utilisés avec GET pour gérer de grands ensembles de données.

PUT : Création et mise à jour	
Définition	La méthode PUT est utilisée pour créer ou mettre à jour une ressource existante.
Idempotence	Elle doit être idempotente, ce qui signifie que répéter une requête PUT avec les mêmes données doit laisser la ressource dans le même état et produire la même réponse que lors de la première demande.
Utilisation	Idéale pour les mises à jour complètes d'une ressource, elle remplace l'état actuel de la ressource par celui fourni.
Sécurité	La mise en œuvre correcte de l'idempotence avec PUT peut nécessiter une gestion soignée des versions de ressources pour éviter les conflits de mise à jour.

Mettre à jour les informations d'un utilisateur

- Idempotence : La méthode PUT assure que la mise à jour peut être répétée sans modifications involontaires.
- Considérations : La gestion des versions peut être nécessaire pour éviter les conflits de mise à jour.

DELETE : Suppression de ressources	
Définition	La méthode DELETE est utilisée pour supprimer une ressource spécifique.

Idempotence	Elle est idempotente, car répéter la requête ne doit pas avoir d'effet supplémentaire une fois la ressource supprimée et ne doit également pas renvoyer une erreur de type 404 NOT FOUND.
Utilisation	Utilisée pour supprimer des ressources, elle doit être maniée avec soin, en particulier dans les systèmes où la suppression a des effets en cascade.
Sécurité	La mise en œuvre de contrôles d'accès appropriés est essentielle pour éviter les suppressions non autorisées.

Supprimer un article de blog

- Idempotence : La méthode DELETE garantit que la suppression peut être répétée sans erreurs supplémentaires.
- Considérations : Les contrôles d'accès et les effets en cascade doivent être soigneusement gérés.

HEAD : Requête d'en-tête	
Définition	La méthode HEAD est utilisée pour récupérer les en-têtes d'une ressource spécifique, sans le corps de la réponse.
Idempotence	Elle doit toujours être idempotente, ce qui signifie que répéter une requête HEAD ne doit pas modifier l'état du système.
Utilisation	Utilisée pour vérifier la disponibilité d'une ressource ou pour vérifier les métadonnées sans télécharger la ressource elle-même.
Sécurité	Comme GET, elle doit être utilisée avec prudence, car les informations d'en-tête peuvent révéler des détails sur la ressource.

Vérifier si un fichier est modifié

- Idempotence : La méthode HEAD garantit que la vérification peut être répétée sans risque.
- Considérations : Utile pour la mise en cache et la validation de la fraîcheur d'une ressource.

OPTIONS : Requête d'options	
Définition	La méthode OPTIONS est utilisée pour décrire les options de communication pour la ressource cible.
Idempotence	Elle doit toujours être idempotente, ce qui signifie que répéter une requête OPTIONS ne doit pas modifier l'état du système.
Utilisation	Utilisée pour déterminer les méthodes HTTP supportées par une ressource ou pour vérifier les exigences de CORS (Cross-Origin Resource Sharing).
Sécurité	Peut révéler des informations sur les capacités du serveur, donc à utiliser avec prudence.

Déterminer les méthodes supportées pour une ressource

- Idempotence : La méthode OPTIONS garantit que la requête peut être répétée sans risque.
- Considérations : Utile pour les clients qui veulent interagir de manière appropriée avec une ressource.

TRACE : Requête de trace	
Définition	La méthode TRACE effectue un test de boucle de retour diagnostique sur la ressource cible.

Idempotence	Elle doit toujours être idempotente, ce qui signifie que répéter une requête TRACE ne doit pas modifier l'état du système.
Utilisation	Utilisée principalement pour le diagnostic, elle permet de voir ce qui est reçu par le serveur, aidant à comprendre les modifications apportées par les proxys.
Sécurité	Peut révéler des informations sensibles et doit donc être utilisée avec prudence.

Tester la boucle de retour d'une ressource

- Idempotence : La méthode TRACE garantit que le test peut être répété sans risque.
- Considérations : Rarement utilisée dans les applications modernes, mais peut être utile pour le débogage et le diagnostic.

Comprendre ces méthodes et savoir comment les appliquer correctement est une compétence essentielle pour tout développeur d'API. Leur utilisation va au-delà de la simple conformité aux spécifications HTTP et s'étend à la conception globale de l'architecture.

3.2 Méthodes potentiellement non-idempotentes

Cette section explore les méthodes qui ne sont pas idempotentes, leurs caractéristiques, et leurs implications dans la conception et l'utilisation d'API.

POST : Création de ressources	
Définition	La méthode POST est généralement utilisée pour créer une nouvelle ressource.
Idempotence	Répéter une requête POST peut générer plusieurs instances de la même ressource, chacune ayant des identifiants distincts attribués par le serveur, ce qui rend cette méthode potentiellement non-idempotente.
Utilisation	Utilisée principalement pour les opérations de création, la méthode POST peut également être appliquée dans des cas spécifiques, comme le traitement de requêtes complexes où les paramètres sont transmis dans le corps de la requête. Dans ces situations, POST peut être utilisé de manière à rester idempotent, par exemple, en réalisant un « GET with body ».
Sécurité	La gestion de l'idempotence avec POST peut nécessiter des mécanismes tels que des jetons d'idempotence[3] pour éviter les créations multiples.

Créer un nouveau commentaire sur un article

- Non-idempotence : La méthode POST peut créer plusieurs instances si elle est répétée.
- Considérations : Les jetons d'idempotence peuvent être utilisés pour éviter les créations multiples.

PATCH : Modifications partielles	
Définition	La méthode PATCH est utilisée pour appliquer des modifications partielles à une ressource existante.
Idempotence	Elle peut être non-idempotente, surtout si les modifications dépendent de l'état actuel de la ressource.
Utilisation	Idéale pour les mises à jour partielles, elle permet de modifier certains attributs d'une ressource sans affecter les autres.
Sécurité	La mise en œuvre de l'idempotence avec PATCH peut être complexe, en particulier lorsque les modifications sont conditionnelles ou basées sur des règles métier.

[3] https://github.com/swiss/api-guidelines?tab=readme-ov-file#may-consider-to-support-idempotency-key-header-230

Modifier le statut d'une commande

- Non-idempotence : La méthode PATCH peut avoir des effets différents si elle est répétée.
- Considérations : La logique métier et les règles de validation doivent être soigneusement conçues.

La compréhension de ces méthodes et de leurs implications est essentielle pour tout développeur d'API, car elle influence non seulement la conception de l'API elle-même, mais aussi la manière dont les clients interagissent avec l'API et gèrent les erreurs et les états.

3.3 Avantages et défis

Les méthodes idempotentes offrent certes des avantages techniques mais présentent par railleurs des défis.

Méthodes idempotentes			
Avantages		**Défis**	
Robustesse	Assurent que les requêtes répétées ont le même effet, augmentant la fiabilité et permettant des reprises sans effets secondaires.	Gestion de l'état	Nécessitent des mécanismes de contrôle de version pour assurer qu'une nouvelle requête produise la même réponse que la première.
Récupération après défaillance	Facilitent la reprise après une défaillance, car les requêtes peuvent être répétées sans risque de modification involontaire de l'état.	Sécurité	Nécessitent des contrôles d'accès pour prévenir les actions non autorisées. Même si certaines méthodes idempotentes peuvent être répétées sans effet secondaire, elles ne diminuent pas les exigences en matière de sécurité.
Simplicité de conception	Réduisent la complexité en éliminant la nécessité de gérer les effets secondaires des requêtes répétées.	Performance	La garantie de l'idempotence peut nécessiter des contrôles supplémentaires, pouvant avoir un impact sur la latence.

Méthodes non-idempotentes			
Avantages		**Défis**	
Flexibilité	Permettent des opérations plus complexes qui dépendent de l'état actuel.	Complexité	La non-idempotence peut augmenter la complexité, nécessitant la gestion des erreurs.
Expressivité	Facilitent l'expression d'opérations qui ne correspondent pas aux autres méthodes HTTP.	Dépendance de l'état	Les méthodes non-idempotentes peuvent dépendre de l'état actuel de la ressource, nécessitant une synchronisation et une gestion d'état.
		Sécurité	La non-idempotence peut exposer des risques de sécurité, nécessitant des contrôles d'accès et des validations supplémentaires.

4 Gestion de la non-idempotence

4.1 Problèmes potentiels

Ce sous-chapitre explore les problèmes techniques pouvant être provoqué par la non-idempotence dans les API, en tentant de fournir des éléments de solutions spécifiques pour chaque cas.

Effets secondaires inattendus

- Problème : Les requêtes répétées peuvent entraîner des modifications involontaires de l'état (ex : créations multiples d'une ressource)
- Solution : Utiliser des jetons idempotents pour identifier de manière unique chaque requête.

Problème potentiel	Solution technique
Créations multiples	Utiliser un jeton unique pour chaque requête POST, vérifier le jeton avant l'exécution.
Modifications involontaires	Utiliser des mécanismes de verrouillage pour éviter les modifications concurrentes.

Incohérence de l'état

- Problème : La non-idempotence peut entraîner une incohérence de l'état entre le client et le serveur, causant des conflits.
- Solution : Mettre en place un contrôle de version et des mécanismes de synchronisation.

Problème potentiel	Solution technique
Conflits d'état	Utiliser des numéros de version ou des horodatages.
Synchronisation inadéquate	Mettre en place des mécanismes de synchronisation.

Effets secondaires inattendus

- Problème : La gestion de la non-idempotence augmente la complexité de l'API.
- Solution : Utiliser des frameworks et des bibliothèques spécifiques qui offrent des mécanismes de gestion de la non-idempotence.

Problème potentiel	Solution technique
Gestion d'erreur complexe	Utiliser des bibliothèques qui fournissent des mécanismes de gestion d'erreur.
Accumulation de jetons	Mettre en place des processus de suppression des jetons expirés.

Problèmes de Performance

- Problème : La gestion de la non-idempotence peut entraîner des problèmes de performance.
- Solution : Optimiser les mécanismes de vérification et utiliser des techniques de mise en cache.

Problème potentiel	Solution technique
Vérification lente des jetons	Utiliser des structures de données optimisées et des techniques de mise en cache pour accélérer la vérification des jetons.
Verrouillage inefficace	Utiliser des techniques de verrouillage pour réduire les contentions et améliorer les performances.

Problèmes de Sécurité

- Problème : La gestion incorrecte de la non-idempotence peut entraîner des problèmes de sécurité (ex : répétition de requêtes malveillantes).

• Solution : Mettre en place des mécanismes de sécurité tels que l'authentification et la validation des jetons.	
Problème potentiel	**Solution technique**
Répétition de requêtes malveillantes	Utiliser l'authentification et la validation des jetons pour empêcher la répétition de requêtes malveillantes.
Exposition de jetons	Utiliser des mécanismes de sécurité pour protéger les jetons idempotents contre l'exposition et l'accès non autorisé.

Problèmes de Compatibilité	
• Problème : La mise en œuvre de la non-idempotence peut entraîner des problèmes de compatibilité avec les clients ou les systèmes plus anciens. • Solution : Fournir des mécanismes de rétrocompatibilité et documenter les comportements non-idempotents.	
Problème potentiel	**Solution technique**
Incompatibilité avec les clients anciens	Fournir des mécanismes de rétrocompatibilité et des alternatives pour les clients qui ne supportent pas les jetons idempotents.
Manque de documentation	Documenter les comportements non-idempotents et les exigences pour les clients et les développeurs.

La gestion de la non-idempotence dans les API est un défi technique qui nécessite une compréhension des problèmes potentiels et des solutions appropriées. Les tableaux et les descriptions fournies dans ce chapitre offrent une piste de réflexion pour les développeurs cherchant à gérer ces problèmes.

4.2 Jetons idempotents

Un jeton idempotent est un identifiant unique associé à une requête API. Il est utilisé pour garantir que les répétitions de cette requête produisent le même résultat sans provoquer d'effets secondaires. En pratique, il permet aux serveurs de reconnaître des requêtes déjà traitées et d'éviter de répéter des opérations.

Cas d'utilisation

Stripe, une plateforme de paiement en ligne, utilise des jetons idempotents pour s'assurer que les requêtes à son API peuvent être répétées sans effet secondaire. Si une requête de paiement échoue en raison d'un problème de réseau, le client peut réessayer la requête avec le même jeton idempotent. Stripe reconnaîtra le jeton et ne traitera pas la requête comme une nouvelle transaction.

4.2.1 Génération de jetons

La génération de jetons idempotents est un processus qui nécessite de considérer les aspects suivants :

- Utiliser des algorithmes cryptographiques forts pour garantir l'unicité et la sécurité des jetons.
- Intégrer les jetons dans les en-têtes ou les corps des requêtes.
- Mettre en place des mécanismes pour détecter et gérer les collisions de jetons.
- En cas de nécessité de réessayer la requête, le client utilise le même jeton idempotent, et le serveur renvoie la même réponse.

Lorsqu'un client souhaite effectuer une opération qui doit être idempotente, il génère un jeton unique, comme un identifiant unique universel (UUID - Universal Unique Identifier). Ce jeton est ensuite inclus dans la requête envoyée au serveur, que ce soit dans l'en-tête, l'URL, ou le corps de la requête.

4.2.2 Vérification de jetons

La vérification des jetons idempotents permet d'assurer que les requêtes non-idempotentes ne sont pas exécutées plusieurs fois.

- Utiliser des techniques de validation cryptographique pour assurer l'intégrité et l'authenticité des jetons.
- Maintenir un état des jetons pour suivre les requêtes associées et prévenir les exécutions multiples.
- Fournir des réponses appropriées en cas de jeton invalide ou expiré, y compris des codes d'état HTTP spécifiques et des messages d'erreur détaillés.

Le serveur vérifie si le jeton a déjà été utilisé pour une opération. Si c'est le cas, il renvoie la réponse précédemment stockée sans effectuer à nouveau l'opération. Si le jeton n'a pas été utilisé auparavant, le serveur traite la requête et stocke le résultat avec le jeton correspondant.

4.2.3 Expiration de jetons

L'expiration des jetons idempotents est un aspect souvent négligé mais vital de leur gestion.

- Définir des politiques d'expiration en fonction des besoins de l'application (ex : durée de vie d'une session utilisateur).
- Mettre en place des mécanismes automatisés de nettoyage des jetons expirés.
- Fournir des mécanismes pour gérer les requêtes associées à des jetons expirés.

Les jetons idempotents peuvent avoir une durée de vie limitée et être supprimés après un certain temps. Dans un système de commerce électronique, par exemple, un jeton idempotent peut être utilisé lors de la création d'une commande pour éviter les doublons en cas de ré-essai de la requête.

4.3 Autres stratégies

Outre les jetons idempotents, il existe d'autres stratégies et techniques pour gérer la non-idempotence dans les API. Le contrôle de version, le verrouillage optimiste, et les réponses détaillées complètent l'utilisation des jetons idempotents.

4.3.1 Contrôle de version

Le contrôle de version est une technique permettant de gérer les conflits.

- Attribuer des numéros de version aux ressources et les incrémenter à chaque modification.
- Mettre en place des règles et des mécanismes pour résoudre les conflits de version.
- Documenter les versions et les changements associés.

4.3.2 Verrouillage optimiste

Le verrouillage optimiste est une technique de contrôle de concurrence qui permet de gérer les accès concurrents sans verrouillage explicite.

- Utiliser des horodatages ou des numéros de version pour détecter les conflits sans verrouiller les ressources.
- Fournir des mécanismes pour détecter et résoudre les conflits de verrouillage.

4.3.3 Réponses détaillées

Fournir des réponses détaillées et informatives est une stratégie qui aide les clients à comprendre et à gérer la non-idempotence.

- Utiliser des codes d'état HTTP appropriés pour indiquer le résultat de la requête non-idempotente.

- Fournir des messages d'erreur détaillés et des informations de débogage.
- Documenter les réponses et les codes d'état dans la documentation de l'API.

5 Directives pour la gestion de l'idempotence dans les API

Ce chapitre fournit des directives pour traiter l'idempotence et compare ces recommandations avec les bonnes pratiques et les lignes directrices de l'industrie, en tenant compte de celles qui ont été établies dans le « Federal Administration API Guidelines »[4], publié ultérieurement. Les tableaux ci-dessous offrent une vue structurée des directives pour la gestion de l'idempotence dans les API. Ils combinent les perspectives générales avec les directives spécifiques, fournissant une référence pratique pour les développeurs.

Utiliser des méthodes idempotentes			
Directive	**Description**	**Exemple**	**Source**
Utiliser les méthodes HTTP appropriées pour les opérations. [GitHub - Méthode HTTP]	GET, HEAD et OPTIONS sont des méthodes sûres, tandis que PUT et DELETE sont des méthodes idempotentes. POST et PATCH ne sont ni sûrs ni idempotents, donc leur utilisation doit être parcimonieuse et prudente.	Utiliser GET pour récupérer des informations, PUT pour mettre à jour une ressource existante, et DELETE pour supprimer une ressource.	Restfull API LinkedIn
Conception de POST et PATCH comme idempotents [GitHub - Idempotence]	Utiliser des clés conditionnelles, secondaires ou d'idempotence	Clé secondaire pour POST	Zalando
Assurer la sécurité des tentatives de nouvelle exécution avec des API idempotentes [GitHub – Idempotence]	Utiliser des opérations idempotentes pour permettre la nouvelle exécution sans effets secondaires supplémentaires.	Utiliser un identifiant de demande client unique pour identifier les demandes en double.	AWS
Gérer les requêtes PUT idempotentes [GitHub – PUT]	Utiliser l'annotation @PutMapping pour les requêtes PUT idempotentes	Utiliser @PutMapping pour s'assurer que la même requête PUT peut être répétée sans changer le résultat	Spring Framework
Utiliser des clés idempotentes [GitHub – Idempotence]	Utiliser une clé unique dans le corps de la requête, l'URL, ou l'en-tête.	Clé unique dans l'en-tête de la requête http (*Idempotency-Key)*	Sofwancoder
Idempotence naturelle [GitHub – Idempotence]	Utiliser des commandes naturellement idempotentes.	« Mettre 1 article dans le panier »	Sofwancoder
Utiliser des liens HATEOAS pour les opérations idempotentes [GitHub – HATEOAS]	En utilisant HATEOAS, on peut guider les clients sur les opérations idempotentes disponibles pour une ressource.	Un lien HATEOAS rel="update" pour une ressource indique qu'une opération PUT (idempotente) est possible.	restfulapi.net

[4] https://github.com/swiss/api-guidelines

Gérer la non-idempotence avec des jetons			
Directive	**Description**	**Exemple**	**Source**
Utiliser des jetons idempotents [GitHub – Idempotence]	Permet aux clients de réessayer en toute sécurité les requêtes en cas d'échec	Réessayer une requête de paiement avec le même jeton idempotent sans double facturation	Stripe API
Eviter POST pour les opérations idempotentes [GitHub – PUT] [GitHub – POST]	POST n'est pas idempotent et peut créer de nouvelles ressources à chaque requête.	Utiliser PUT pour mettre à jour une ressource.	Restfull API
Utiliser une clé secondaire [GitHub – secondary key]	Eliminer les problèmes de ressources en double	ID du panier d'achat dans une ressource de commande	Zalando
Gérer les demandes en retard et les changements de paramètres dans les appels idempotents [GitHub – Idempotence]	Gérer les cas où les demandes arrivent en retard ou où les paramètres de la demande changent.	Retourner une erreur de validation si les paramètres d'une demande idempotente ne correspondent pas à une demande précédente avec le même identifiant.	AWS
Utiliser des codes d'état, des en-têtes et un corps appropriés pour indiquer le résultat des requêtes. [GitHub – HTTP status codes]	Eviter les effets secondaires ou les dépendances cachées qui peuvent affecter d'autres ressources ou composants.	Utiliser le code d'état 200 pour une réussite, 404 pour une ressource non trouvée, et fournir des messages d'erreur détaillés.	LinkedIn
Middleware pour l'idempotence [GitHub – Idempotence]	Utiliser un middleware pour intercepter et gérer les requêtes idempotentes.	Middleware vérifiant la clé idempotente.	Sofwancoder
Cache et base de données [GitHub – Cache]	Utiliser un cache ou la base de données pour suivre les requêtes traitées.	Utilisation de Redis pour le suivi des requêtes.	Sofwancoder
Utiliser des liens HATEOAS pour les opérations non-idempotentes [GitHub – POST] [GitHub – HATEOAS]	HATEOAS peut aider à indiquer clairement quand une opération non-idempotente, comme POST, est appropriée.	Un lien HATEOAS rel="create" suggère une action POST pour ajouter une nouvelle ressource.	restfulapi.net

Fournir des réponses appropriées			
Directive	**Description**	**Exemple**	**Source**
Utiliser des codes d'état http [GitHub – HTTP status codes]	Aider les clients à comprendre le résultat	Code d'état 409 pour une requête POST répétée	Général

Soutenir l'en-tête Idempotency-Key [GitHub – idempotency-key]	Assurer un comportement idempotent fort	Stocker la clé unique de la requête temporairement	Zalando
Assurer la consistance des réponses dans les appels idempotents [GitHub – semantic]	Fournir des réponses sémantiquement équivalentes pour les demandes idempotentes.	Retourner la même réponse pour une demande répétée avec le même identifiant de demande client, même si la ressource a été supprimée entre les demandes.	AWS
Atomicité	Assurer l'atomicité entre l'écriture de l'ID et les changements.	Transaction atomique en base de données.	Sofwancoder
Utiliser des liens HATEOAS pour guider les réponses [GitHub – HATEOAS]	Les liens HATEOAS peuvent guider le client sur les actions possibles après une requête, en fonction du résultat.	Après une opération DELETE réussie, un lien HATEOAS rel="restore" pourrait être fourni pour une éventuelle restauration.	restfulapi.net

Construction de fonctions idempotentes dans un contexte de Cloud			
Directive	**Description**	**Exemple**	**Source**
Utilisation des identifiants d'événements [GitHub – UUID]	Utiliser des identifiants uniques pour chaque événement pour assurer l'idempotence.	Peut être utilisé avec des bases de données distribuées.	Google Cloud
Utilisation des clés d'idempotence [GitHub – idempotency-key]	Utiliser des clés uniques pour chaque requête pour garantir que les requêtes répétées ont le même effet.	Utilisé dans les systèmes de paiement et autres systèmes critiques.	Google Cloud
Mécanisme de bail	Utiliser un mécanisme de bail pour éliminer l'exécution parallèle de sections non idempotentes.	Utile dans les systèmes distribués où l'ordre d'exécution peut varier.	Google Cloud
Consistance [GitHub – idempotency-key]	Garantir la consistance entre différents systèmes.	Synchronisation entre microservices.	Sofwancoder
Performance et latence [GitHub – Performance]	Gérer la latence introduite par la gestion de l'idempotence.	Optimisation des requêtes de base de données.	Sofwancoder
Utiliser HATEOAS pour la découverte dynamique [GitHub – HATEOAS]	Dans un environnement cloud, les ressources peuvent être éphémères. HATEOAS permet une découverte dynamique des opérations idempotentes disponibles.	Si une instance de service est déplacée ou redémarrée, HATEOAS peut fournir des liens mis à jour pour les opérations idempotentes.	restfulapi.net

Documenter l'idempotence			
Directive	**Description**	**Exemple**	**Source**
Documenter les comportements idempotents [GitHub – idempotence]	Aider les développeurs à utiliser l'API correctement	Section dans la documentation de l'API	Général
Concevoir pour le traitement hors séquence idempotent	Permettre des systèmes extrêmement résilients	Envoyer l'identifiant du processus/ressource/entité	Zalando
Documenter les contrats idempotents et les attentes en matière de nouvelle exécution	Documenter clairement les contrats idempotents et les attentes en matière de nouvelle exécution.	Expliquer comment les identifiants de demande client uniques sont utilisés et quelles sont les attentes en matière de nouvelle exécution.	AWS
Utiliser DELETE avec l'identifiant de la ressource [GitHub – DELETE]	DELETE est idempotent avec l'identifiant de la ressource, mais pas sans.	DELETE /item/{id} est idempotent.	Restfull API
Tester et surveiller l'idempotence et la sécurité dans les API REST	Utiliser des outils et des techniques pour simuler différents scénarios et vérifier les résultats attendus. Utiliser des outils comme Postman, Rest Assured, ou SoapUI pour créer et exécuter des cas de test. Utiliser des outils comme Prometheus, Grafana, ou Elastic pour collecter des métriques et des logs.	Créer des cas de test avec Postman pour envoyer les mêmes requêtes à vos points de terminaison API REST et vérifier les réponses.	LinkedIn
Documenter les clés idempotentes [GitHub – idempotency-key]	Documenter l'utilisation de clés idempotentes dans l'API.	Documentation Swagger pour les clés idempotentes.	Sofwancoder
Intégrer HATEOAS dans la documentation de l'API [GitHub – HATEOAS]	La documentation de l'API doit clairement montrer comment HATEOAS est utilisé pour indiquer les opérations idempotentes.	Dans la documentation, montrer comment un client peut suivre un lien HATEOAS rel="update" pour effectuer une mise à jour idempotente.	restfulapi.net

6 Défis actuels

Le tableau ci-dessous fournit une vue d'ensemble des défis techniques liés à l'idempotence dans les API. Il couvre divers aspects, y compris la standardisation, la sécurité, l'interopérabilité, la performance, la formation, le monitoring, et la testabilité, en fournissant des descriptions spécifiques et des solutions techniques pour chaque défi.

Thème	Description	Solution technique
Standardisation et conformité	• La diversité des protocoles et des normes peut entraîner des incohérences dans l'implémentation de l'idempotence. • La mise en place de mécanismes de validation pour assurer la conformité aux directives d'idempotence.	Utilisation de frameworks et de bibliothèques standardisés, développement de tests automatisés pour valider la conformité.
Sécurité et confidentialité	• Risques de réutilisation malveillante des jetons idempotents. • Nécessité de protéger l'intégrité des jetons tout en assurant leur fonction de détection des requêtes répétées.	Utilisation de mécanismes de validation des jetons idempotents pour éviter les réutilisations abusives, tels que la mise en cache temporaire des jetons et la surveillance des tentatives de duplication.
Performance et scalabilité	• Assurer la performance sous haute charge. • Gérer efficacement les identifiants uniques pour les requêtes. • Utilisation optimale des mécanismes de caching.	Utilisation de bases de données performantes, solutions de caching comme Redis, architecture scalable.
Formation et sensibilisation	• Nécessité de former les développeurs aux principes d'idempotence. • Création et maintenance de documentation.	Organisation de workshops, formations, utilisation d'outils de documentation comme Swagger.
Monitoring et logging	• Mise en place de mécanismes de suivi pour les requêtes idempotentes. • Analyse des logs pour détecter les problèmes.	Utilisation d'outils de monitoring comme Prometheus, solutions de logging comme ELK Stack.
Testabilité	• Création de tests pour assurer une couverture complète. • Assurer que tous les aspects de l'idempotence sont testés.	Utilisation de frameworks de test comme JUnit ou Mocha, mise en place de pipelines de CI/CD.

7 Références

- https://fr.wikipedia.org/wiki/Idempotence
- https://blog.sofwancoder.com/idempotency-in-apis-planning-for-uncertainty
- https://fr.wikipedia.org/wiki/Alg%C3%A8bre_de_Boole_(logique)
- https://datatracker.ietf.org/doc/html/rfc7231
- https://stripe.com/docs/api/idempotent_requests
- https://docs.aws.amazon.com/fr_fr/amazondynamodb/latest/developerguide/DynamoDBContext.VersionSupport.html
- https://nordicapis.com/understanding-idempotency-and-safety-in-api-design/
- https://docs.spring.io/spring-framework/reference/
- https://opensource.zalando.com/restful-api-guidelines/
- https://github.com/microsoft/api-guidelines/blob/vNext/azure/Guidelines.md
- https://aws.amazon.com/fr/builders-library/making-retries-safe-with-idempotent-APIs/
- https://cloud.google.com/blog/products/serverless/cloud-functions-pro-tips-building-idempotent-functions?hl=en
- https://restfulapi.net/idempotent-rest-apis/
- https://www.linkedin.com/advice/0/how-do-you-test-monitor-idempotency#:~:text=One%20of%20the%20best%20practices,use%20them%20sparingly%20and%20carefully.
- https://martinfowler.com/articles/richardsonMaturityModel.html
- swiss/api-guidelines: Federal Administration API Guidelines (github.com)

Annexe A - HATEOAS (Hypermedia as the Engine of Application State)

HATEOAS est un principe de l'architecture REST visant à rendre les API auto-découvrables. Dans le contexte de l'Administration fédérale l'adoption de HATEOAS peut grandement améliorer l'interopérabilité des services.

Exemple concret

Imaginons une API de gestion des ressources humaines au sein de l'Administration fédérale. Lorsqu'un utilisateur récupère les détails d'un employé, l'API pourrait fournir des liens vers ses formations, ses affectations précédentes, ou même vers des ressources pour demander des congés ou des formations supplémentaires.

L'idempotence et HATEOAS sont deux concepts distincts mais complémentaires dans le contexte des API REST. Voici comment ils sont liés :

Aspect	Description	Détail technique	Exemple
Navigation et actions idempotentes	HATEOAS fournit des liens vers des actions possibles. Si ces actions sont idempotentes, les clients peuvent répéter les requêtes sans craindre des effets secondaires.	Lorsqu'un client suit un lien HATEOAS vers des méthodes idempotentes, il peut être assuré que répéter la requête n'aura pas d'effet secondaire, grâce à la nature idempotente de ces méthodes.	Un citoyen vérifiant le statut de sa demande de document officiel via un lien HATEOAS peut cliquer plusieurs fois sans affecter le statut réel de la demande.
Sécurité et fiabilité	L'idempotence garantit que le système reste dans un état cohérent, même en cas de problèmes réseau ou de réessais.	Les en-têtes HTTP, tels que *Idempotency-Key*, peuvent être utilisés pour garantir l'idempotence des requêtes, en particulier pour les méthodes non-idempotentes comme POST.	Si un citoyen soumet une demande de document et que la requête est interrompue en raison d'une défaillance réseau, il peut la réessayer en utilisant la même *Idempotency-Key* pour s'assurer qu'il ne crée pas de demandes en double.
Simplicité pour les clients	Les clients peuvent suivre les liens HATEOAS sans avoir à connaître tous les endpoints à l'avance, ce qui facilite la navigation dans l'API. En parallèle, l'idempotence des méthodes HTTP permet de répéter des requêtes sans causer d'effets secondaires.	Les clients n'ont pas besoin de mémoriser les endpoints. Ils peuvent simplement suivre les liens fournis dans les réponses, en s'assurant que les méthodes idempotentes ne modifient pas l'état du système lors des ré-essais	Un citoyen utilisant une application pour accéder à des services gouvernementaux n'a pas besoin de connaître tous les endpoints ; il peut simplement suivre les liens fournis pour accomplir différentes tâches.
Evolution des API	HATEOAS permet d'introduire de nouvelles actions sans perturber les clients existants. L'idempotence garantit que les anciens clients peuvent toujours interagir de manière fiable.	Lors de l'ajout de nouveaux liens ou actions via HATEOAS, s'assurer que ces actions respectent les principes d'idempotence. Cela garantit la compatibilité ascendante.	Si de nouveaux services sont ajoutés à l'Administration fédérale, ils peuvent être intégrés à l'API existante via HATEOAS, garantissant que les applications existantes continuent de

			fonctionner sans modification.

Avantages de HATEOAS pour l'Administration fédérale :

- Les utilisateurs ou les systèmes peuvent naviguer dans l'API sans avoir besoin d'une documentation externe.
- Les changements dans l'API, tels que l'ajout de nouvelles fonctionnalités, peuvent être introduits sans casser les clients existants.
- Les clients peuvent interagir avec l'API en suivant les liens fournis, sans avoir besoin de construire des URL ou de connaître à l'avance les endpoints.

Modèle de maturité de Richardson pour les API REST

Le modèle de maturité de Richardson, conçu par Leonard Richardson, est un cadre qui évalue la maturité des services web en fonction de leur adhérence aux principes REST. Il décrit la progression d'une API depuis l'utilisation du protocole HTTP jusqu'à l'adoption des contraintes REST, en particulier HATEOAS (Hypertext as the Engine of Application State).

Ce modèle offre une feuille de route pour les développeurs souhaitant créer des API REST. Il permet une forme d'auto-documentation, où les clients peuvent comprendre comment interagir avec l'API sans se référer à une documentation externe. Cela facilite l'intégration et l'interopérabilité, en particulier dans des contextes complexes comme celui de l'Administration fédérale.

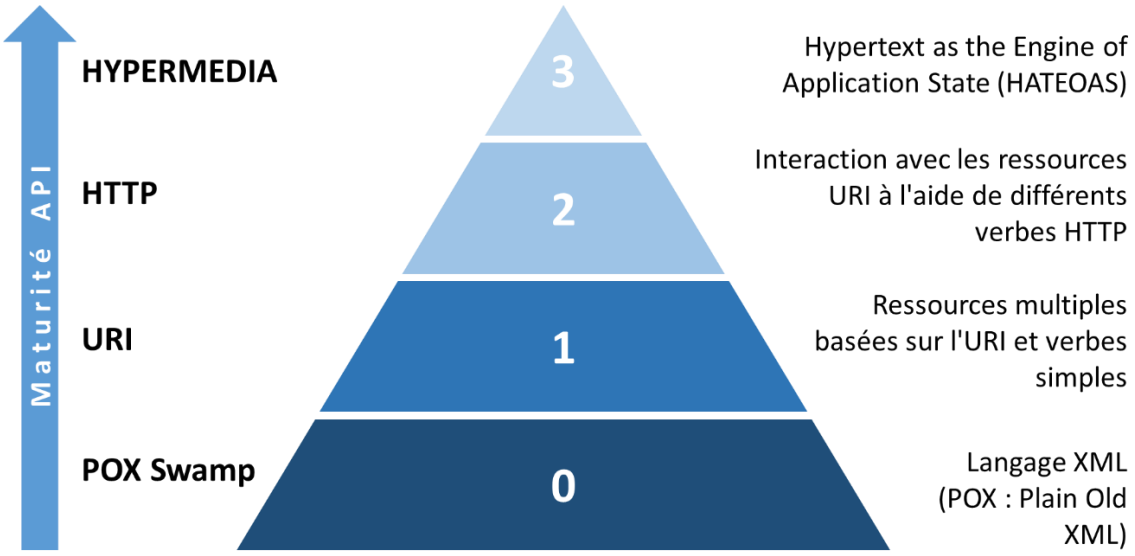

Figure 2 : Modèle de maturité de Richardson

Niveau	Directive	Description	Méthode HTTP	Caractéristique
Niveau 0 POX Swamp	Utilisation de HTTP comme tunnel	Utilisation d'HTTP comme simple protocole de transport pour des appels RPC.	POST	Un seul point d'entrée (URI).
Niveau 1 Ressources	Introduction des ressources	Les ressources individuelles sont identifiées avec des URI distinctes.	POST	Chaque ressource a sa propre URI.

Niveau	Directive	Description	Méthode HTTP	Caractéristique
Niveau 2 Verbes HTTP	Utilisation des verbes HTTP	Utilisation des méthodes HTTP pour interagir avec les ressources.	GET POST PUT DELETE	Les réponses utilisent des codes de statut HTTP appropriés.
Niveau 3 HATEOAS	Contrôles hypermédia (HATEOAS)	L'API est pleinement RESTful avec l'intégration de HATEOAS.	GET POST PUT DELETE	Les réponses fournissent des liens vers d'autres ressources.

- **Niveau 0** : À ce stade, l'API est centrée sur l'utilisation d'opérations distantes. Les services qui restent à ce niveau ont tendance à être fortement couplés.

- **Niveau 1** : La distinction des URI pour chaque ressource permet une meilleure organisation et une identification claire des éléments sur lesquels les opérations sont effectuées.

- **Niveau 2** : L'utilisation des méthodes HTTP standardise l'interaction avec les ressources. Par exemple, l'utilisation de GET pour récupérer des données, de POST pour créer une nouvelle ressource, de PUT pour la mise à jour et de DELETE pour la suppression. Cette standardisation facilite la compréhension et l'utilisation de l'API.

- **Niveau 3** : À ce niveau, l'API devient auto-découvrable. Les clients peuvent naviguer dans l'API simplement en suivant les liens fournis dans les réponses, sans avoir besoin d'une connaissance préalable de la structure de l'API.

Des informations intéressantes sur le principe de fonctionnement de ce modèle peuvent être obtenues sur le lien suivant : https://martinfowler.com/articles/richardsonMaturityModel.html#level3