Rapport – Laboratoire 3 SLH

Auteur

- Harun Ouweis

1. Réponses aux Questions

1. Y a-t-il des problèmes avec la politique spécifiée dans l'énoncé ?

La politique définie présente plusieurs zones d'ombres et potentiels risques :

- **Pouvoir absolu des administrateurs** : Leur accorder des droits illimités sans restrictions spécifiques peut poser problème en matière de sécurité et d'auditabilité. Une approche plus fine, telle que l'ajout d'une validation par un second administrateur (principe de double validation) pour les opérations sensibles, pourrait être envisagée.
- **Absence de gestion post-relation médecin-patient** : Rien n'est précisé quant à la fin de la relation entre un médecin et un patient. Peut-il encore accéder à l'historique médical après avoir cessé d'être son médecin traitant ? Une clarification de cette règle serait nécessaire.
- **Manque d'un accès d'urgence** : Dans un cadre médical, des médecins en situation d'urgence doivent parfois accéder aux dossiers de patients qu'ils ne traitent pas habituellement. L'absence d'une telle exception dans la politique pourrait poser problème en cas de soins critiques.

En résumé, bien que fonctionnelle, la politique pourrait être raffinée pour mieux cadrer certains scénarios critiques en matière de sécurité et d'accès exceptionnel.

---

2. Quelles politiques seraient difficiles à implémenter avec un modèle RBAC au lieu d'ABAC ?

Un modèle RBAC (Role-Based Access Control) repose uniquement sur des rôles prédéfinis, ce qui rend complexe l'implémentation de politiques dépendant du contexte. Parmi celles définies, plusieurs seraient particulièrement contraignantes à gérer sans ABAC :

- **L'accès aux données personnelles par l'utilisateur lui-même** : En RBAC, un rôle "Utilisateur" aurait les mêmes droits pour tous, alors qu'ici, chaque utilisateur doit pouvoir lire et modifier uniquement son propre dossier.
- **L'accès d'un médecin aux données de ses patients** : Il faudrait gérer dynamiquement la relation médecin-patient avec des rôles intermédiaires, alors qu'en ABAC, cette relation est simplement définie par un attribut (`medical_folder.doctors`).
- **L'auteur d'un rapport pouvant consulter et modifier ses propres écrits** : Un rôle "Médecin" ne pourrait pas différencier son propre rapport de ceux d'autres médecins, sans une logique externe ajoutée au RBAC.

En RBAC pur, ces politiques nécessiteraient de multiples rôles ou des exceptions codées en dur, ce qui alourdirait la gestion des permissions et rendrait la maintenance du système plus complexe. ABAC offre ici une flexibilité précieuse.

Voici une **vérification détaillée** des réponses de ton collègue avant de formuler une réponse claire et distincte pour ton rapport :

---

3. Que pensez-vous de l'utilisation d'un Option mutable dans la structure Service pour garder trace de l'utilisateur loggué ?

L'utilisation d'une `Option<UserID>` mutable pour suivre l'utilisateur connecté présente certaines limites. Elle oblige à vérifier l'authentification à chaque appel de méthode, rendant le code plus complexe et augmentant le risque d'erreurs d'exécution.

Une alternative plus robuste serait de tirer parti du système de types de Rust en distinguant deux structures de service distinctes :

- `UnauthenticatedService` qui ne permet que les actions accessibles sans connexion.
- `AuthenticatedService` qui contient un `user_id` et donne accès aux actions nécessitant une authentification.

Cette approche permet de détecter les erreurs à la compilation, réduisant ainsi le besoin de vérifications répétées à l'exécution.

Dans un contexte multi-utilisateur (API web), conserver l'authentification dans le service peut poser problème si plusieurs requêtes accèdent simultanément à la même instance. Une meilleure solution serait de passer l'`user_id` en paramètre à chaque appel ou de gérer les sessions utilisateur séparément.

---

4. Que pensez-vous de l'utilisation de la macro de dérivation automatique pour Deserialize pour les types de model et de input_validation ?

- **Pour les types de modèle (`models`)** : La dérivation automatique de `Deserialize` est généralement adaptée, car ces structures correspondent souvent directement aux données sérialisées. Toutefois, il est essentiel d'être attentif aux champs sensibles, comme les mots de passe ou les données médicales, qui ne doivent pas être exposés involontairement.

- **Pour la validation des entrées (`input_validation`)** : L'usage de `#[derive(Deserialize)]` est moins pertinent, car il permet d'instancier directement des objets sans vérifier la validité des données. Une approche plus sûre consiste à implémenter une logique de désérialisation personnalisée avec `TryFrom` ou `Deserialize`, garantissant ainsi qu'aucun objet invalide n'est créé.

En résumé, la dérivation automatique est utile pour simplifier la sérialisation des modèles, mais une validation stricte est préférable pour les entrées utilisateur afin d'éviter des erreurs en aval.

---

5. Quel est l'impact de l'utilisation de Casbin sur la performance de l'application et sur l'efficacité du système de types ?

Impact sur les performances

Casbin fonctionne en évaluant dynamiquement les règles de politique, ce qui introduit une surcharge au moment de l'exécution.

- Pour un faible volume de requêtes, l'impact est négligeable, mais lorsque le nombre d'opérations augmente, les temps de traitement peuvent devenir significatifs.
- Des solutions existent pour limiter cet impact, notamment l'utilisation d'un cache des décisions fréquentes, afin d'éviter une réévaluation systématique des mêmes règles.
- Des tests ont montré que la vérification exhaustive de toutes les combinaisons possibles (utilisateurs, actions, ressources) pouvait prendre plusieurs minutes. L'optimisation des scénarios testés a permis de conserver un temps d'exécution raisonnable.

Impact sur le système de types

Casbin applique les règles uniquement à l'exécution, ce qui crée une rupture avec le système de typage fort de Rust.

- Contrairement aux permissions gérées via le type system, une erreur dans la politique (ex. une règle mal définie) ne sera détectée qu'à l'exécution, augmentant le risque d'oublis ou d'incohérences.
- L'éditeur n'offre pas d'auto-complétion pour la rédaction des règles Casbin, ce qui rend leur développement plus délicat.
- Pour garantir la cohérence des règles, il est nécessaire de tester les politiques de manière approfondie afin d'identifier d'éventuelles erreurs de configuration.

---

6. Avez-vous d'autres remarques ?

Pas vraiment