

Faculté de Génie Département de génie électrique et de génie informatique

**Système de gestion des salles de cours**

Présenté par

**Philippe Garneau garp2405**

**Alexandre Gagnon gaga2515**

**11 SEPTEMBRE 2018**

# INTRODUCTION

Dans le cadre des cours GIF-620 et GIF-390, Système de gestion des salles de cours à l’Université de Sherbrooke, notre mandat était de gérer les salles de cours de la faculté de génie à l’aide d’une architecture solide de base de données. Pour ce faire, il nous fallait tout d’abord penser à la conception de notre base de données : minimiser les périodes d’interruption, faciliter la maintenance, être transparent, modulable…

Par la suite, la modélisation de notre système de façon conceptuelle ainsi que relationnelle était nécessaire pour générer la base de notre code SQL permettant de créer nos différentes tables ainsi que leurs relations. Par la suite, la complexité de notre situation nous a demandé d’intégrer des triggers ainsi que des procédures pour bien gérer les différents événements possibles ainsi que pour garder un historique des différentes actions faites par l’utilisateur. Pour compléter le tout, il nous a été demandé de s’interroger sur la notion d’entrepôt de données et si notre application en est une.

**DIAGRAMME DE DÉPLOIEMENT**

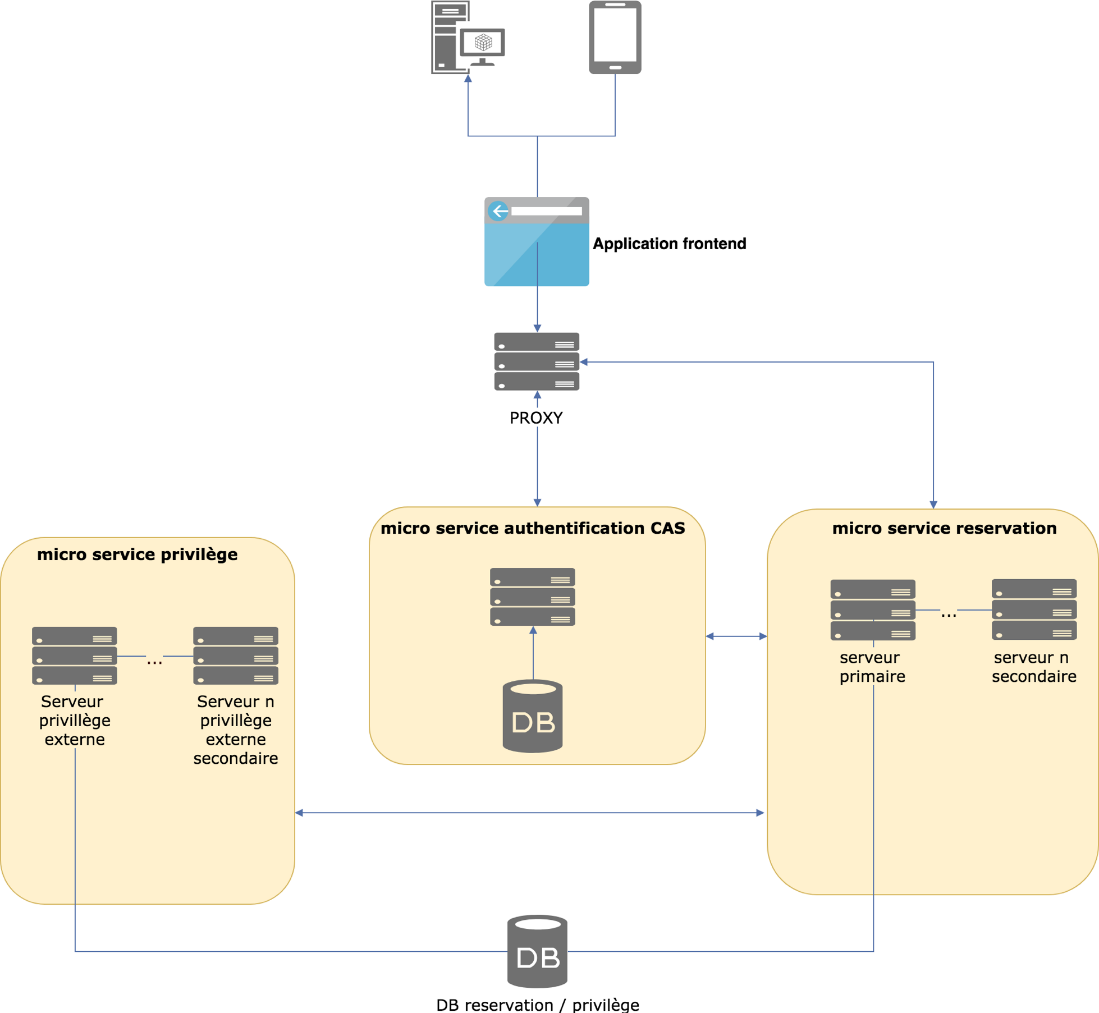


Figure 1Diagramme de déploiement

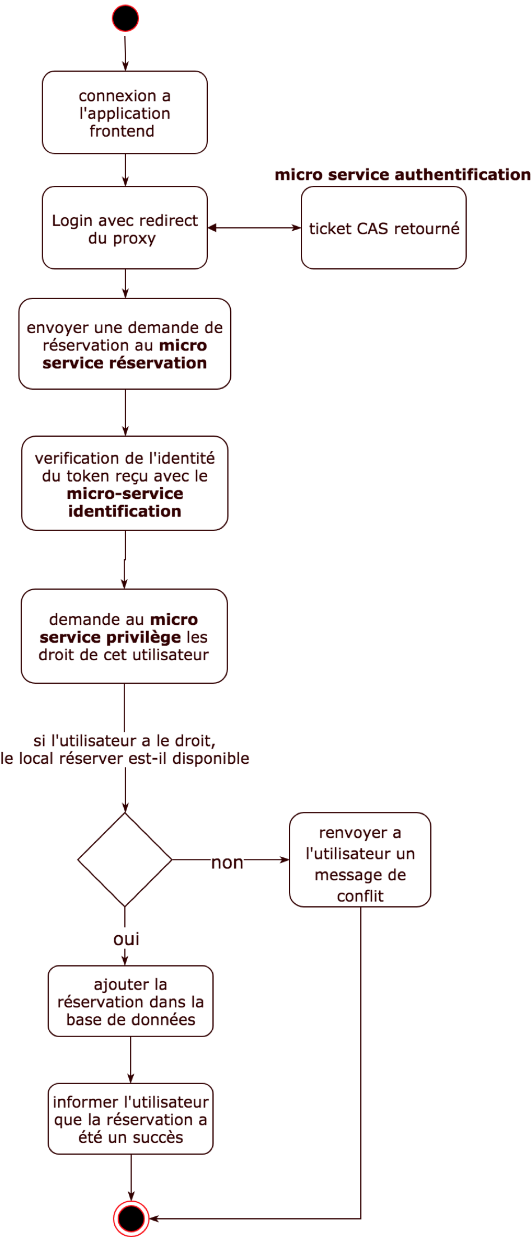
**MODÈLES D’ARCHITECTURE UTILISÉS**

Comme préciser dans les lignes directrices du guide étudiant, nous avons ici une application multi-plate-forme protégée en accès par un proxy utilisant le serveur d’authentification CAS. Ce proxy, donnant à notre application une plus grande transparence, sert à rediriger le trafic vers les micros services approprié. Ce proxy nous permet aussi de confirmer la connexion aux serveurs secondaires dans le cas de panne ou de maintenance assurant ainsi des périodes d’interruption minimale. Le nombre de serveurs secondaires est variable dépendant des moyens ainsi que des besoins de l’application. Dans notre cas, comme il s’agit seulement de quelques milliers d’étudiants accédant a une plage horaire de locaux, il serait suffisant d’avoir 1 seul serveur secondaire.

Pour ce qui est de l’architecture de notre application, nous avons décidé d’utiliser une architecture microservice. C’est un choix qui nous permet de développer de nouvelles fonctionnalités plus facilement et rapidement sur les microservices individuels. Il y a aussi une meilleure séparation des « concerns », donc chaque microservice est plus simple à développer s’il fait une tâche très spécifique et simple. Par contre, un des désavantages majeurs d’une architecture microservice est la latence plus grande dû au nombre plus élevé de requêtes http par action de l’utilisateur.

En ayant une application ayant la totalité de sa logique présente dans ses microservices, l’application client a simplement besoin de pouvoir faire des requêtes http vers notre backend, donc cela permet une très grande hétérogénéité. L’utilisation d’un proxy ou d’un load balancer nous donne une très forte transparence, car ces composants s’occupent de relayer l’information envoyée par l’utilisateur aux bons microservices. Aussi, il n’y a aucune reconstruction d’information à performer au niveau de l’application client, car ce rôle est réservé au microservice de réservation pour l’instant. Dans un futur lointain, nous pourrions implémenter ce qu’on appelle un « Backend For Frontend » qui peut s’occuper de faire la combinaison des différentes réponses de divers microservices. L’avantage du design BFF est qu’un BFF peut être créé par plateforme, donc on s’assure que l’application reste hétérogène. Connaissant que notre application utilise une architecture microservice, il y a un très faible couplage entre nos différents microservices et l’application client.

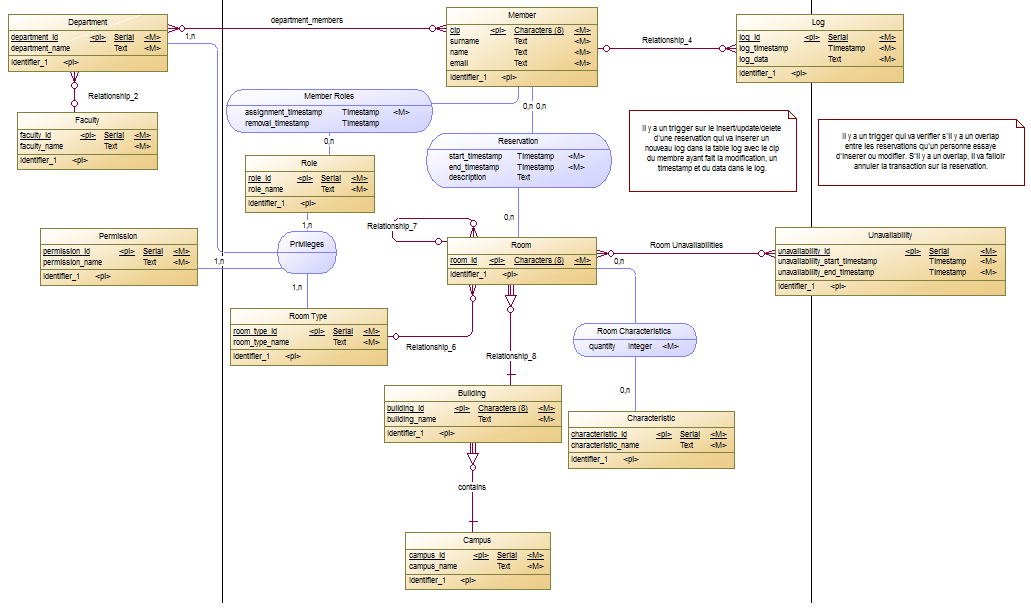
Le diagramme d’activité ci-dessous nous démontre les étapes à effectuer pour se connecter puis réserver un local.



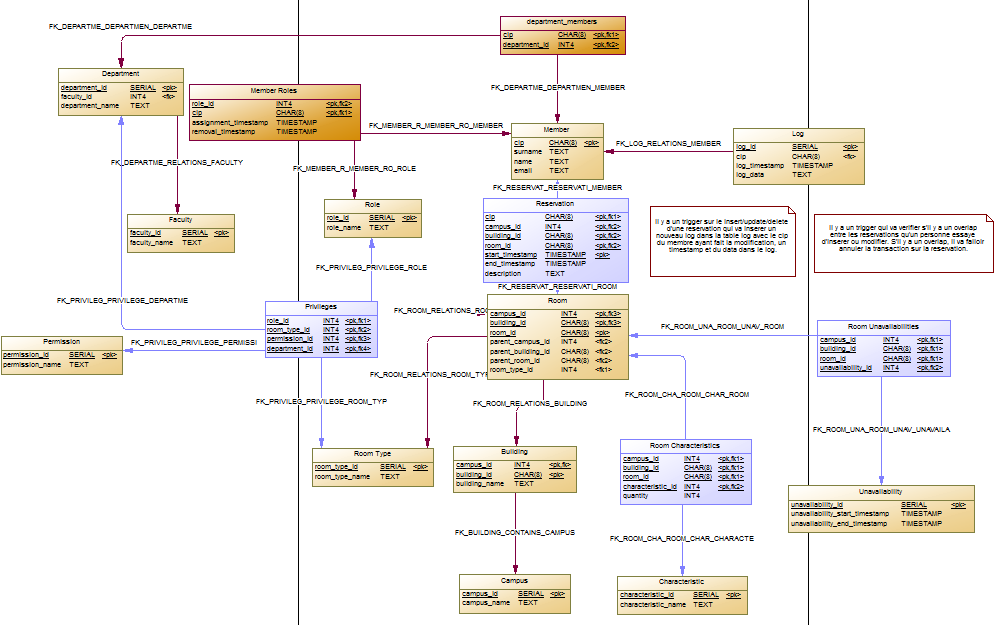
Premièrement, l’application frontend, dans notre cas une application web, redirige la requête d’authentification de l’utilisateur vers le CAS par un proxy et le CAS répond à celle-ci avec un ticket encrypté contenant l’identification de l’utilisateur.

Par la suite, l’utilisateur envoie une requête pour créer une réservation qui est redirigée par le proxy. Le service de réservation vérifie l’identité contenue dans la requête avec le CAS et ensuite vérifie avec le service de privilèges pour s’assurer que cet utilisateur à le droit de faire une réservation. Si tout est en règle, la réservation est ajoutée et l’utilisateur reçoit une réponse indiquant le succès de son opération.

**MODÈLE CONCEPTUEL**



**MODÈLE RELATIONNEL**



Notre modèle relationnel respecte la 1ère forme normale pour les raisons suivantes :

1. L’information contenue dans les cellules est indivisible, donc l’atomicité est respectée.

Notre modèle relationnel respecte la 2ème forme normale pour les raisons suivantes :

1. Le modèle respecte la 1ère forme normale
2. Tous les attributs non-clés des tables sont dépendant de l’entièreté de la clé primaire composée.

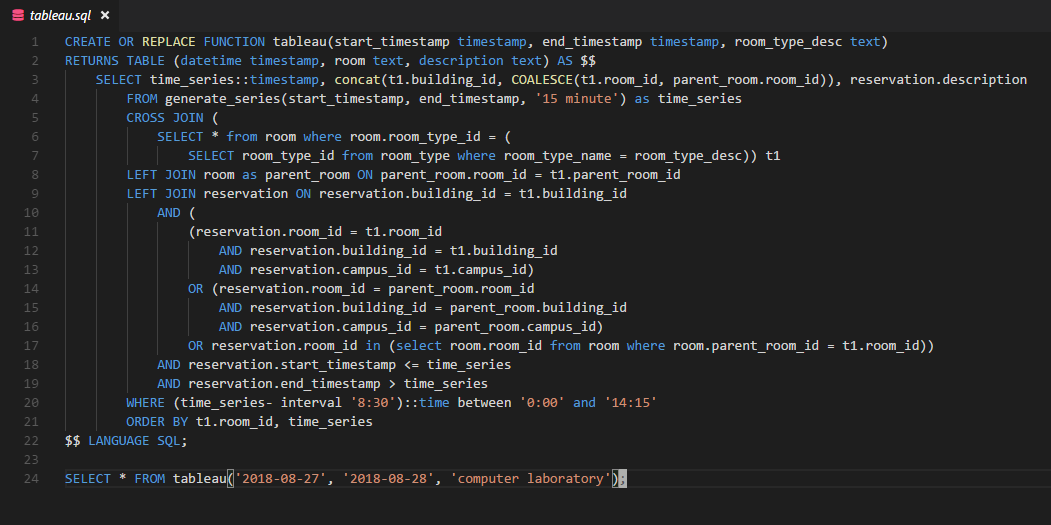
Notre modèle relationnel respecte la 3ème forme normale pour les raisons suivantes :

1. Le modèle respecte la 2ème forme normale
2. Tous les attributs non-clés des tables n’ont aucune dépendance transitive avec d’autres éléments non clés de la table.

Voici les modifications manuelles apportées à notre modèle relationnel après sa génération automatique :

1. Ajout de l’attribut start\_timestamp dans la clé primaire de la table Reservation. Ce changement était nécessaire, car sinon, une personne n’aurait pu réserver la même salle à un autre moment. Avec l’ajout du start\_timestamp, ce problème est réglé.
2. Modification du nom de FK2 dans la table Room, car ces foreign keys permettent de créer le lien entre les cubicules et leur local parent. Le nom automatiquement généré ne représentait pas très bien le rôle de ces foreign keys.

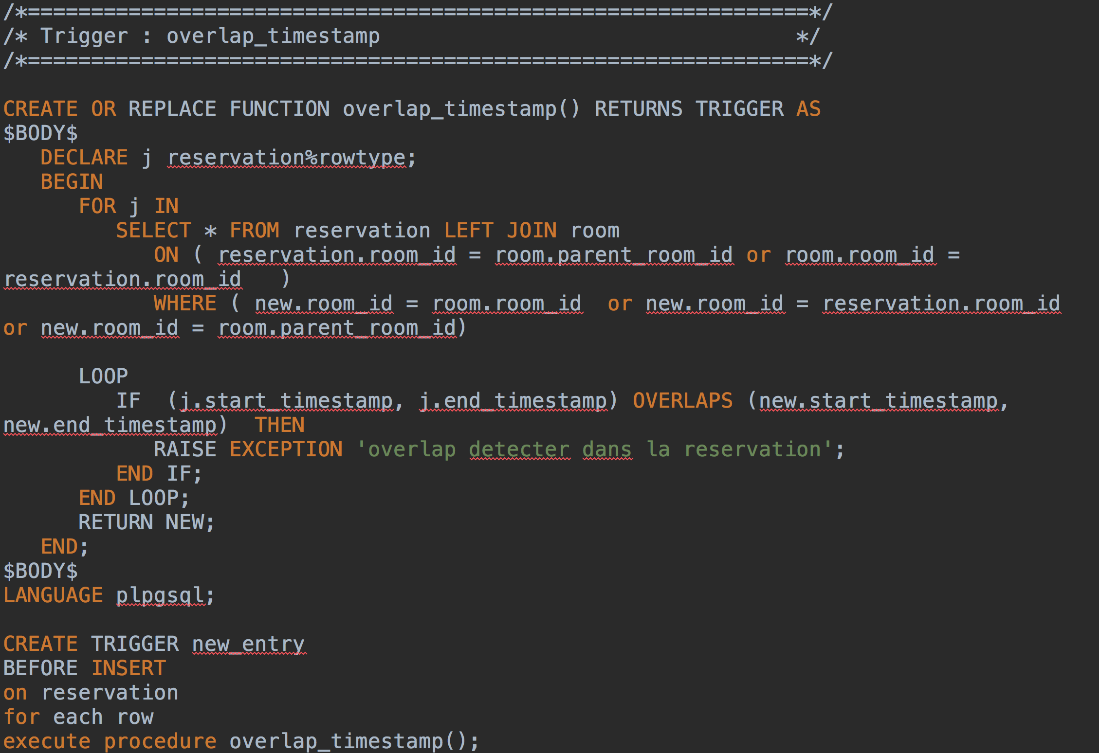
**PROCÉDURE DU TABLEAU DE RÉSERVATION INCLUANT LA GESTIONS DES CUBICULES**



# PROBLÈMES DE LA LISTE DES LOCAUX

Il n’y a aucun problème avec la liste des caractéristiques et la liste des fonctions. Cependant, la liste des locaux pour réservation est un amas d’information qui ne devrait pas nécessairement vivre ensemble. Premièrement, la première forme normale n’est pas respectée par les colonnes « Nom du local » et « Liste des caractéristiques ». Le nom du local est une combinaison de l’identifiant unique de pavillon dans lequel le local se trouve ainsi qu’un identifiant unique du local à l’intérieur de ce pavillon. Cette information devrait être séparée en deux colonnes pour former une clé primaire composée. La colonne « List des caractéristiques » contient des cellules qui ont plus d’un élément d’information et cela ne respecte pas le principe d’atomicité. Pour corriger ce problème, il faudrait une autre table qui aura le code du pavillon, le code du local et le code de la caractéristique comme clé primaire composée. Cela permet de faire le lien entre un seul local unique et ses multiples caractéristiques. La colonne « Notes » n’apporte pas grand-chose en termes d’information supplémentaire présentement, mais je peux voir une utilité à cette colonne avec de l’information pertinente qui ne pourrait être modélisée par la table faisant le lien entre le local et ses caractéristiques.

# GESTION DU CHEVAUCHEMENT DES RÉSERVATIONS

Pour les chevauchements de réservation, nous avons opté pour un design basé sur un trigger étant appelé avant l’insertion d’une rangée. Le trigger est le suivant :

Donc, à chaque fois qu’une nouvelle entrée est ‘insert’, la procédure overlap\_timestamp() est appelée. Comme nos nouvelles valeurs sont stockées dans la valeur new, il nous est facile de comparé l’intervalle de celles-ci (début,fin) aux intervalles de la table réservation. La fonction OVERLAPS nous permet de faire cette comparaison plutôt facilement, car elle nous ramène un booléen positif si chevauchement il y a. Comme passer sur toute notre table de réservation est très peu intéressant et qu’il faut regarder si une certaine salle à un parent ou bien un enfant, nous avons tout d’abord effectué un LEFT JOIN sur la table

réservation et room. Cette décision vient du fait que notre vérification comporte 3 niveau de: les autres réservations de la même salle, les autres réservations de la salle parente et les autres réservations des cubicules enfants. Pour simplifier celle-ci, nous avons utilisé le left joint sur les rangé ayant le room\_id de réservation comme parent et sur les room ayant le même room id que les réservations. De cette façon, on voit dans une table les entrées des autres cubicules (si cubicule il y a) ainsi que les parents (si parent il y a) de la salle. On compare par la suite à l’aide de la FOR loop puis on évalue le overlap contre chaque entité sélectionnée.

Cette manière, quoique très efficace, n’est pas optimale pour un système qui se voudrait bien plus volumineux que celui de l’université. Une manière plus simple et moins demandant, qui pourrait être développée dans une autre itération, serait de simplement faire un joint en ajoutant les bonnes conditions nous ramenant seulement les locaux de même type étant en chevauchement. Cela nous permet d’empêcher un looping non nécessaire et de simplement générer une erreur si le joint ramène un résultat.

# ENTREPÔT DE DONNÉES - INDEXATION

Un entrepôt de données est une base de données qui est supposée sauvegarder l’information qui pourra être utilisée dans le futur pour la prise de décisions d’une entreprise grâce aux statistiques extraites de cette information. Ce sont des données qui ne sont pas supposées être manipulées plusieurs fois par jour. La base de données que nous avons conçue n’est pas un entrepôt de données, car elle sauvegarde de l’information très volatile qui changera possiblement plusieurs fois par jour et cela à tout moment durant l’année. Cependant, il pourrait être intéressant de mettre l’information entourant les réservations dans un entrepôt de données, car beaucoup d’information utile pourrait être extraite de l’historique des réservations. En revanche, je crois que quelques tables devraient être modifiées pour ne pas nécessairement respecter la troisième forme normale pour rendre l’information plus centralisée dans l’entrepôt de données.

Les deux types de colonnes les plus susceptibles d’être indexées sont des colonnes qui sont souvent utilisées dans des WHERE and celles utilisées dans les différents types de JOIN. Les colonnes utilisées dans des clauses WHERE sont souvent utilisées pour rechercher une valeur très spécifique, donc l’indexation est importante pour accélérer ces requêtes. Le même principe s’applique aux joints, car il y a souvent des comparaisons de valeurs spécifiques dans des colonnes. Il peut être utile aussi de créer des index composés lorsque deux colonnes sont souvent utilisées ensemble pour faire des recherches. Dans notre cas, les colonnes les plus susceptibles d’être utilisées le plus souvent sont : building\_name, characteristic\_name, room\_type\_name, role\_name, permission\_name, department\_name. Ce sont sous des attributs qui seront très souvent utilisés dans les clauses WHERE, car ils sont tous très fortement reliés aux réservations et aux locaux qui sont les deux tables qui seront les plus sollicitées par notre application.

# CONCLUSION

Lors de l’implémentation et la mise en œuvre de notre système, nous avons rencontré plusieurs risques. Le manque d’expérience avec le langage SQL unique à PostgreSQL ainsi que le code plpgsql est un immense risque qui pourrait engendrer des erreurs et une performance non optimale. Le fait que nous ayons beaucoup de logique présente dans la database et non dans l’application nous empêche de faire une procédure de test rigoureuse pour nos triggers et fonctions.

Honnêtement, je ne crois pas qu’il y ait des éléments d’un système de réservation qui ne soient pas couverts pas notre conception. Il est évident que nous aurions pu élargir le type de choses pouvant être réservées comme des livres, outils, etc, mais ce problème pourrait simplement être réglé en ajoutant des tables représentant ces nouveaux types d’item et ajouter de nouveaux privilèges.