Fast and flexible NIST level 2 hierarchical RBAC with jRBAC

Nowadays enterprise authorization is a tedious task, both for the enterprise developer and the administrator. A flexible and performance critical authorization system, Specifically a u Based Access Control mechanism, would be what many enterprises might benefit. This document discusses a NIST hierarchical RBAC implementation with many features.

Keywords: *jFramework, RBAC, SQL*

Abbas Naderi
Shahid Beheshti University
abiusx@acm.org

1. Introduction

Role based access control is a very flexible pattern of authorization, and has been used in a variety of enterprises under a lot of stress. Due to its separation of users and permissions, flexibility is at hand.

In a RBAC system, there are three entities, a *user* as the actor of the system, one or more *roles* for every user, and a few *permissions* assigned to every role. The most common operation of a RBAC system would be to check a permission against a user for authorization.

With growing enterprises, either on the external size (which is the physical size of the organization) and internal size (as the complexity of enterprise applications) more roles are necessary for the system, hence many more permissions. Management of a lot of roles and permissions, and both user/role and role/permission relations, Would be no easy task for an administrator, if ever possible. For example an organization with 500 users, 1000 roles and 50,000 permissions would most likely have around 1,000,000 role/permission relations which would require huge resources for maintenance (and in most scenarios would result in unsafe configuration and many leaks). Thus, hierarchical RBAC is defined to outwit the problem via hierarchies, either as inheritance or aggregation.

jFramework which is a PHP web application framework developed around simplicity and extensibility, implements a very fast and flexible RBAC subsystem, namely jRBAC. This article intends to cover implementation details criteria of this subsystem.

2. The Hierarchy

NIST RBAC standard consists of four incremental levels. First level known as *Flat RBAC* is not satisfactory for large scale enterprises as noted in previous section. Level 2 RBAC, named *hierarchical RBAC*, indicates that only roles need to be hierarchical, either in a tree/inverted tree or any hierarchical structure.

Most implementations of hierarchical RBACs usually deploy trees (or inverted trees), since they are more identical to the true world organizational structures, and are

much easier to manage both for the developer and the administrator.

The key point missed in the NIST standard is that roles are not very frequent (since they're in correspondence with people and people usually don't take many roles, each) yet permissions (each for an action or task) are numerous. Due to the nature of this problem, it would've been much wiser to force *permission hierarchies* with role hierarchy as an optional benefit. jRBAC implements both roles and permissions in hierarchies in a simple and intelligent manner.

3. Dependencies

jRBAC is based on RDBMS and SQL, since they are very commonly used in nowadays platforms and applications. One might note that SQLs are not the preferable choice for embedded devices and low level applications due to the requirement of a RDBMS server setup on the platform. The solution lies in SQLite, the almost new database technology that resides in only a single file and requires no database server but a tiny driver deployed with the application. jRBAC has been tested on both SQLite and MySQL in many operational situations and prevailed (more discussion at Benchmark section)

The current version of jRBAC is implemented with PHP programming language, but has but little PHP code and most of its code are compatible SQL statements, so it can be ported to any desired language with almost no effort. Also the implementation is part of the jFramework PHP framework (www.jframework.info) but only uses framework's platform for database queries (which can be easily replaced to achieve independency).

For a flexible and performance critical implementation of hierarchies, The *Nested Set* hierarchical model is used, which is a means of managing hierarchical data fast and easy via RDBMS tabular structure. In a Nested Set, every row has two metadata fields namely *Left* and *Right* instead of a parent key. These fields are filled respectively based on the pre-order and post-order traversal of the actual tree. The modeling is wise since it allows us to directly know children and descendants of a node as well as perform batched operations on all of them in a single SQL query.

4. Modeling

To see an organization in the eyes of a RBAC subsystem, we should present a simple modeling.

The first entity would be a *User* which is equal to a virtual/real person inside the organization. The user is considered authentic and intends to perform some tasks inside the system, So RBAC should decide if the user has the permission for it or not, Hence the second entity would be a *Permission* which represents a ticket required to do a single action.

Since permissions are not related to users directly, but on their role in the organization, there comes a *Role* which is a node in the organization structure. As in real world situations, A user might have one or more roles assigned to him and a role might be assigned to one or more users. Users change frequently (in compare with roles and permissions) but roles remains quite the same over time, with some additions.

A role has a set of permissions to do a set of certain tasks. For example Role A might have permissions to perform action M, N and role B might have permissions to perform actions N, O and P. The set of role/permission assignments also remains the same over time in the organization (until a new role or permissions is defined)

Since roles are hierarchical, if some user has a role which is an ancestor of another role, he/she would have the union of all descendants' permissions. This is useful for defining a general role with almost no permissions, and defining sub-roles with appropriate permissions of all. The more general role would be the logical supervisor of all the sub-roles, thus having all their permissions.

Permissions are also hierarchical, but in and inverse manner, i.e a general action is defined as a permission, and it's then divided into more atomic actions. If someone has the permissions to do the general action, he/she also has the permission to do all the atomic sub-actions which are descendants of the more general task. This behavior helps both define actions/permissions and assign them to roles, e.g you won't need to assign all related atomic permissions to a role, instead you assign their parent action to the desired role.

The tabular representation of entities is as simple as possible, yet not lacking basic features, So we have ID, Title for roles and permissions and ID, Username for users. We also have 2 MxN relation tables one for user/role relations and the other for role/permission relations both as simple as possible (with a unique pair of foreign keys).

The Roles table and Permissions table both have have *Left, Right* fields for an effective use of nested sets as well.

This modeling should be kept in mind for the following sections of the article, since both role and permission hierarchies could act differently (inheritance instead of specialization)

5. Operations

5.1 Check

The most frequently desired operation of a RBAC system is to check a permission against a user, called *Check()* from now on. To check against a particular user, we need to perform the following sub operations:

1. Acquire all direct roles of the user
2. Union all indirect roles of the user - which are descendants of the direct roles - with the acquired direct roles.
3. Acquire all permissions assigned to each role in the set of roles from step 2.
4. Union all the acquired permissions
5. Find all indirect permissions of the user (which are ascendants of permissions from step 4 inclusive)
6. Check the final list of permissions resulted from step 5 against the desired permission.

This method is the first one that comes to mind, But obviously not the best since the set of items we have to keep in memory increases exponentially by each step and we search for a needle in a haystack at step 6, which is definitely unwise. The following method achieves the same goal with both fewer and simpler steps:

1. Acquire all direct roles of the user
2. Union all indirect roles of the user - which are descendants of the direct roles - with the acquired direct roles.
3. Acquire all ascendant permissions of our desired permission (inclusive)
4. Check to see if any of the roles in step 2 is assigned to any of the permissions in step 3.

As one might notice, we union all descendants of user roles to know all the roles and sub-roles the user can act as, but we union all ascendants of the desired permission since if the user has permission for any of them, He/she has the permission to perform the desired action.

The trick lies in Meet-In-The-Middle (MITM) concept, which significantly reduces computational order of some NP algorithms.

The following SQL query performs the whole algorithm in a single fast non-correlative query against the database with Username and Permission Title as inputs :

```
SELECT COUNT(*) AS Result FROM
```

```
  `TableUsers` AS TU
JOIN `TableUserRoles` AS TUR ON (TU.`UserID`=TUR.`UserID`)

JOIN `TableRoles` AS TRdirect ON (TRdirect.`RoleID`=TUR.`RoleID`)
        JOIN `TableRoles` AS TR
        ON ( TR.`Left` BETWEEN TRdirect.`Left` AND TRdirect.`Right`)
JOIN
(
  `TablePermissions` AS TPdirect
  JOIN `TablePermissions` AS TP
        ON ( TPdirect.`Left` BETWEEN TP.`Left` AND TP.`Right`)
  JOIN `TableRolePermissions` AS TRP
        ON (TP.`PermissionID`=TRP.`PermissionID`)
) ON (TR.`RoleID` = TRP.`RoleID`)
WHERE
  TU.`Username`= {1}
AND
  TPdirect.`Title` = {2}
```

It is quite obvious that we require to join all five tables for a Check(), but there are two more extra joins. These joins both employ goals of the algorithm and ensure a non-correlative query.

As described by the algorithm, first all direct roles are acquired via joining Users, User/Roles and Roles tables. Secondly, All indirect roles are derived. Afterwards, All indirect permissions packed (the parenthesis section) and lastly, we check which of these permissions are in assignment with those roles by joining both with Role/Permissions table by the condition of assignment.

The first extra join is applied on roles table again, causing the resulting temporary table to hold all indirect roles (i.e roles that are descendant of direct roles inclusive), as well.

The second extra join (employed inside the parenthesis) is applied on the permissions table again, to acquire all ascendants of direct permissions (i.e indirect permissions). As it can be seen, This time the join condition is "all permissions that direct permissions are between them", instead of when we acquired descendant roles with the join condition "all roles that are between direct roles".

Lastly, WHERE conditions of the desired username and [direct] permission title are applied to end the query.

The result is the number of paths our user reaches the desired permission. If more than zero, the permission must be considered granted and otherwise, access is denied.

5.2 RoleHasPermission

Sometimes -mostly in a virtual manner- the subsystem is required to tell if a role has access to a particular permission or not. This scenario happens either when tuning Role/Permission assignments or when batching interfaces to present based on roles.

This operation is quite similar to the operation done in Check(), but is employed otherwise to be more related to Nested Set concepts and practices. In this query, the necessary tables are joined and conditions are applied with subqueries:

```
SELECT COUNT(*) AS Result FROM
 `TableRoles` AS TR
 JOIN `TableRolePermissions` AS TRP ON (TR.`RoleID`=TRP.`RoleID`)
 JOIN `TablePermissions` AS TP ON ( TP.`PermissionID` = TRP.`PermissionID`)
```

```
WHERE
  TR.`Left` BETWEEN
  (SELECT `Left` FROM `TableRoles` WHERE `Title` = {1} )
  AND
  (SELECT `Right` FROM `TableRoles` WHERE `Title` = {1} )
AND
  TP.`PermissionID` IN
  (
     SELECT Parent.`PermissionID`
     FROM     `TablePermissions` AS Node,
              `TablePermissions` AS Parent
     WHERE Node.`Left` BETWEEN Parent.`Left` AND Parent.`Right`
              AND Node.`Title`= {2}
     ORDER BY Parent.`Left`
  );
```

This operation performs much faster than Check() with less memory consumption, since for every distinct role a user has, Check() requires all its descendants to be acquired - which forms an exponential growth in number of available paths - but this operation only needs descendants of a particular role to be gathered. That is why two wholly different approaches of SQL query are used for these two similar operations.

5.3 UserInRole

The third operation highly desired in a RBAC subsystem is to check if a user has a role or not, and is straightforward. Administrators often would like to present some reports or any piece of information to a batch of users under a certain role. Also reports and statistics are generated based on roles a user has performed. In these two category of scenarios, Two different operations are required to determine if a user is in a role or not. The former includes indirect user roles as well, but the latter needs only direct user roles.

Since we've already discussed two role descendants scenario, the former UserInRole case is omitted and the latter done as follows:

```
SELECT COUNT(*) AS Result FROM
`TableUserRoles` WHERE `UserID`= {1}
and `RoleID` = {2}
```

If username and role title are presented, we simply join those two tables as well.

6. Benchmark

7. Further Work

8. Conclusion

9. References

MySQL Dev : Managing hierarchical data in MySQL (Nested Sets) http://dev.mysql.com/tech-resources/articles/hierarchical-data.html

jRBAC Benchmark : Raw Results, Charts, Tests and Environments http://wiki.jframework.info/index.php?title=RBAC_Benchmark