hsadmin-ng's Role-Based-Access-Management (RBAC)

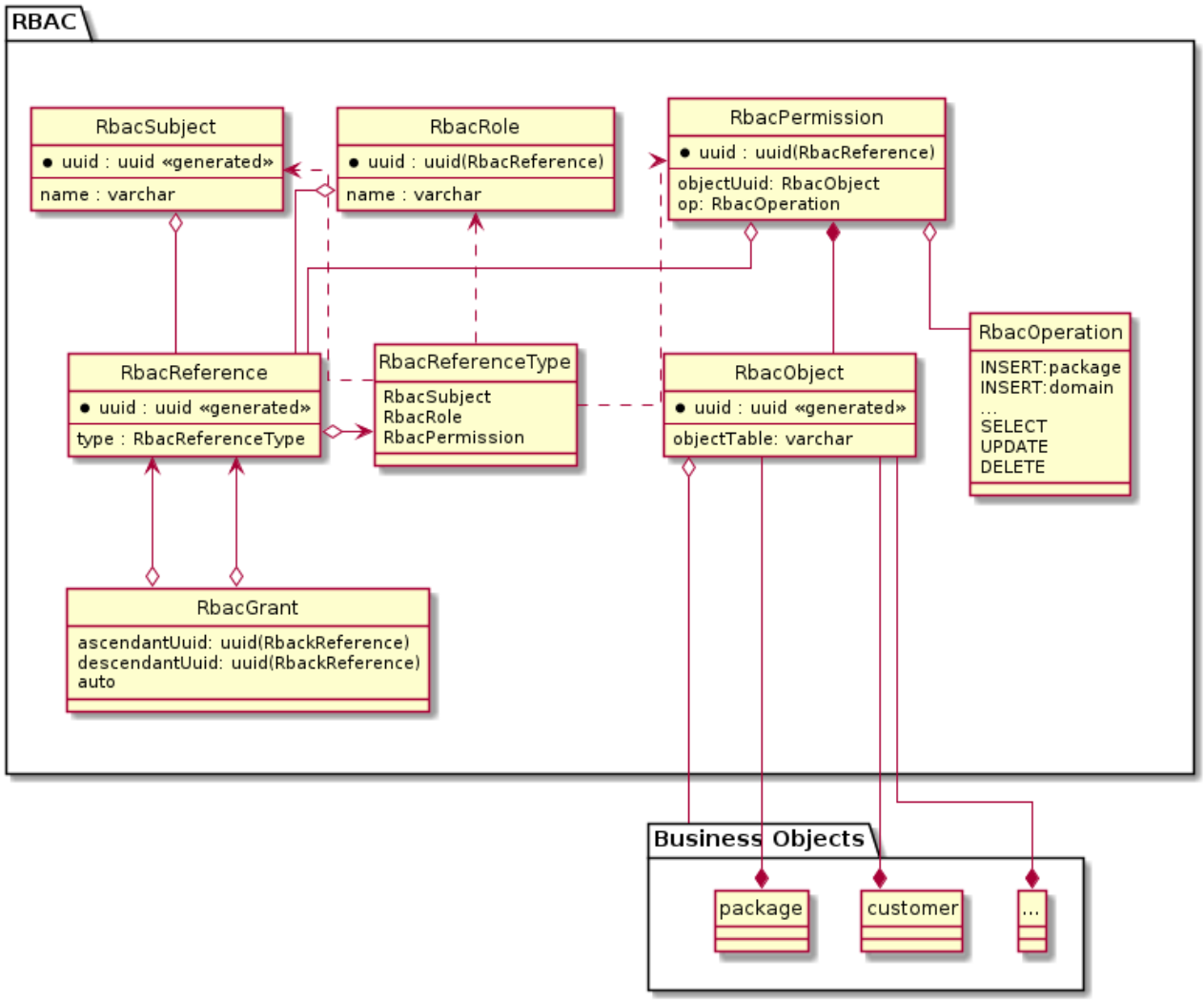
The requirements of *hsadmin-ng* include table-, row- and column-level-security for read and write access to business-objects. More precisely, any access has to be controlled according to given rules depending on the accessing users, their roles and the accessed business-object. Further, roles and business-objects are hierarchical.

To avoid misunderstandings, we are using the term "business-object" what's usually called a "domain-object". But as we are in the context of a webhosting infrastructure provider, "domain" would have a double meaning.

Our implementation is based on Role-Based-Access-Management (RBAC) in conjunction with views and triggers on the business-objects. As far as possible, we are using the same terms as defined in the RBAC standard, for our function names though, we chose more expressive names.

In RBAC, subjects can be assigned to roles, roles can be hierarchical and eventually have assigned permissions. A permission allows a specific operation (e.g. SELECT or UPDATE) on a specific (business-) object.

You can find the entity structure as a UML class diagram as follows:



The RBAC Entity Types

RbacReference An *RbacReference* is a generalization of all entity types which participate in the hierarchical role system, defined via *RbacGrant*.

The primary key of the *RbacReference* and its referred object is always identical.

RbacReferenceType The enum *RbacReferenceType* describes the type of reference. It's only needed to make it easier to find the referred object in *RbacSubject*, *RbacRole* or *RbacPermission*.

RbacSubject An *RbacSubject* is a type of RBAC-subject which references a login account outside this system, identified by a name (usually an email-address).

*RbacSubject*s can be assigned to multiple *RbacRole*s, through which they can get permissions to *RbacObject*s.

The primary key of the *RbacSubject* is identical to its related *RbacReference*.

RbacRole An *RbacRole* represents a collection of directly or indirectly assigned *RbacPermission*s. Each *RbacRole* can be assigned to *RbacSubject*s or to another *RbacRole*.

Both kinds of assignments are represented via *RbacGrant*.

RbacRole entities can *RbacObject*s, or more precise

RbacPermission An *RbacPermission* allows a specific *RbacOperation* on a specific *RbacObject*.

RbacOperation An *RbacOperation* determines, what an *RbacPermission* allows to do. It can be one of:

- **'INSERT'** - permits inserting new rows related to the row, to which the permission belongs, in the table which is specified an extra column, includes 'SELECT'
- **'SELECT'** - permits selecting the row specified by the permission, is included in all other permissions
- **'UPDATE'** - permits updating (only the updatable columns of) the row specified by the permission, includes 'SELECT'
- **'DELETE'** - permits deleting the row specified by the permission, includes 'SELECT'

This list is extensible according to the needs of the access rule system.

Please notice, that there is no **create** operation to create new instances of unrelated business-object-types. For such a singleton business-object-type, e.g. *Organization" or "Hostsharing" has to be defined, and its single entity is referred in the permission. Only with this rule, the foreign key in* RbacPermission* can be defined as `NOT NULL`.

RbacGrant The *RbacGrant* entities represent the access-rights structure from *RbacSubject*s via hierarchical *RbacRoles* down to *RbacPermission*s.

The core SQL queries to determine access rights are all recursive queries on the *RbacGrant* table.

Role naming

The naming pattern of a role is important to be able to address specific roles. E.g. if a new package is added, the admin-role of the related customer has to be addressed.

There can be global roles like 'administrators'. Most roles, though, are specific for certain business-objects and automatically generated as such:

```
business-object-table#business-object-name.role-stereotype
```

Where *business-object-table* is the name of the SQL table of the business object (e.g *customer* or 'package'), *business-object-name* is generated from an immutable business key(e.g. a prefix like 'xyz' or 'xyz00') and the *role-stereotype* describes a role relative to a referenced business-object as follows:

owner The owner-role is granted to the subject which created the business object. E.g. for a new *customer* it would be granted to 'administrators' and for a new *package* to the 'customer#...:ADMIN'.

Whoever has the owner-role assigned can do everything with the related business-object, including deleting (or deactivating) it.

In most cases, the permissions to other operations than 'DELETE' are granted through the 'admin' role. By this, all roles ob sub-objects, which are assigned to the 'admin' role, are also granted to the 'owner'.

ADMIN The admin-role is granted to a role of those subjects who manage the business object. E.g. a 'package' is manged by the admin of the customer.

Whoever has the admin-role assigned, can usually update the related business-object but not delete (or deactivating) it.

The admin-role also comprises lesser roles, through which the SELECT-permission is granted.

AGENT The agent-role is not used in the examples of this document, because it's for more complex cases. It's usually granted to those roles and users who represent the related business-object, but are not allowed to update it.

Other than the tenant-role, it usually offers broader visibility of sub-business-objects (joined entities). E.g. a package-admin is allowed to see the related debitor-business-object, but not its banking data.

TENANT The tenant-role is granted to everybody who needs to be able to select the business-object and (probably some) related business-objects. Usually all owners, admins and tenants of sub-objects get this role granted.

Some business-objects only have very limited data directly in the main business-object and store more sensitive data in special sub-objects (e.g. 'customer-details') to which tenants of sub-objects of the main-object (e.g. package admins) do not get SELECT permission.

GUEST (Deprecated)

REFERRER Like the agent-role, the guest-role too is not used in the examples of this document, because it's for more complex cases.

If the referrer-role exists, the SELECT-permission is granted to it, instead of to the tenant-role. Other than the tenant-role, the referrer-roles does never grant any roles of related objects.

Also, if the referrer-role exists, the tenant-role receives the SELECT-permission through the referrer-role.

Referenced Business Objects and Role-Depreciation

A general rule is, if one business object *origin* references another object *target* (in other words: one database table joins another table), **and** a role for *origin* needs also access to *target*, then usually the *target* role is granted to the *origin* role which is one level lower.

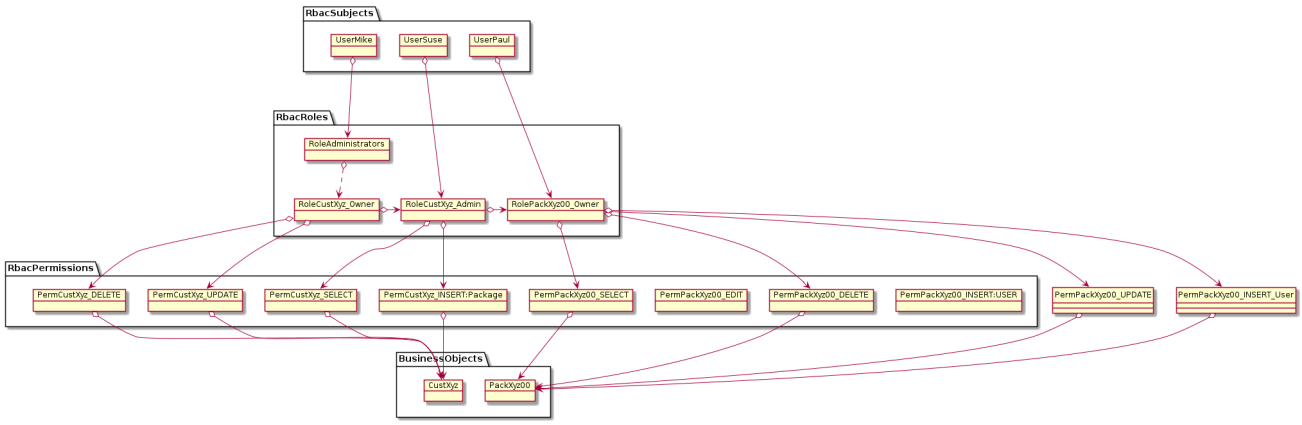
E.g. the admin-role of the *origin* object gets granted the agent-role (or, if it does not exist, then the tenant-role) of the *target* object.

Following this rule, also implies, that the number of indirections to which visibility can be granted is limited. The admin-role of one object could be granted visibility to another object through at maximum 3 joins (agent->tenant->guest).

But not in all cases role-depreciation takes place. E.g. often a tenant-role is granted another tenant-role, because it should be again allowed to select sub-objects. The same for the agent-role, often it is granted another agent-role.

Example Users, Roles, Permissions and Business-Objects

The following diagram shows how users, roles and permissions could be granted access to operations on business objects.



Business-Object-Tables, Triggers and Views

To support the RBAC system, for each business-object-table, some more artifacts are created in the database:

- a `BEFORE INSERT TRIGGER` which creates the related *RbacObject* instance,

- an `AFTER INSERT TRIGGER` which creates the related *RbacRole*s, *RbacPermission*s together with their related *RbacReference*s as well as *RbacGrant*s,
- a restricted view (e.g. *customer_rv*) through which restricted users can access the underlying data.

Not yet implemented, but planned are these actions:

- an `ON DELETE ... DO INSTEAD` rule to allow `SQL DELETE` if applicable for the business-object-table and the user has 'DELETE' permission,
- an `ON UPDATE ... DO INSTEAD` rule to allow `SQL UPDATE` if the user has 'UPDATE' right,
- an `ON INSERT ... DO INSTEAD` rule to allow `SQL INSERT` if the user has the 'INSERT' right for the parent-business-object.

The restricted view takes the current user from a session property and applies the hierarchy of its roles all the way down to the permissions related to the respective business-object-table. This way, each user can only select the data they have 'SELECT'-permission for, only create those they have 'add-…'-permission, only update those they have 'UPDATE'- and only delete those they have 'DELETE'-permission to.

Current User

The current use is taken from the session variable `hsadminng.currentSubject` which contains the name of the user as stored in the *RbacSubject*s table. Example:

```
SET LOCAL hsadminng.currentSubject = 'mike@hostsharing.net';
```

That user is also used for historicization and audit log, but which is a different topic.

Assuming Roles

If the session variable `hsadminng.assumedRoles` is set to a non-empty value, its content is interpreted as a list of semicolon-separated role names. Example:

```
SET LOCAL hsadminng.assumedRoles = 'customer#aab:admin;customer#aac:admin';
```

In this case, not the current user but the assumed roles are used as a starting point for any further queries. Roles which are not granted to the current user, directly or indirectly, cannot be assumed.

Example

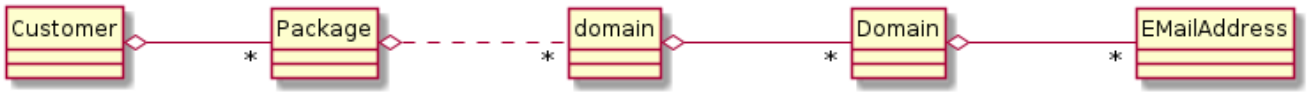
A full example is shown here:

```
BEGIN TRANSACTION;
    SET SESSION SESSION AUTHORIZATION restricted;
    SET LOCAL hsadminng.currentSubject = 'mike@hostsharing.net';
    SET LOCAL hsadminng.assumedRoles = 'customer#aab:admin;customer#aac:admin';

    SELECT c.prefix, p.name as "package", ema.localPart || '@' || dom.name as "email-address"
      FROM emailaddress_rv ema
      JOIN domain_rv dom ON dom.uuid = ema.domainuuid
      JOIN domain_rv uu ON uu.uuid = dom.domainuuid
      JOIN package_rv p ON p.uuid = uu.packageuuid
      JOIN customer_rv c ON c.uuid = p.customeruuid;
END TRANSACTION;
```

Roles and Their Assignments for Certain Business Objects

To give you an overview of the business-object-types for the following role-examples, check this diagram:

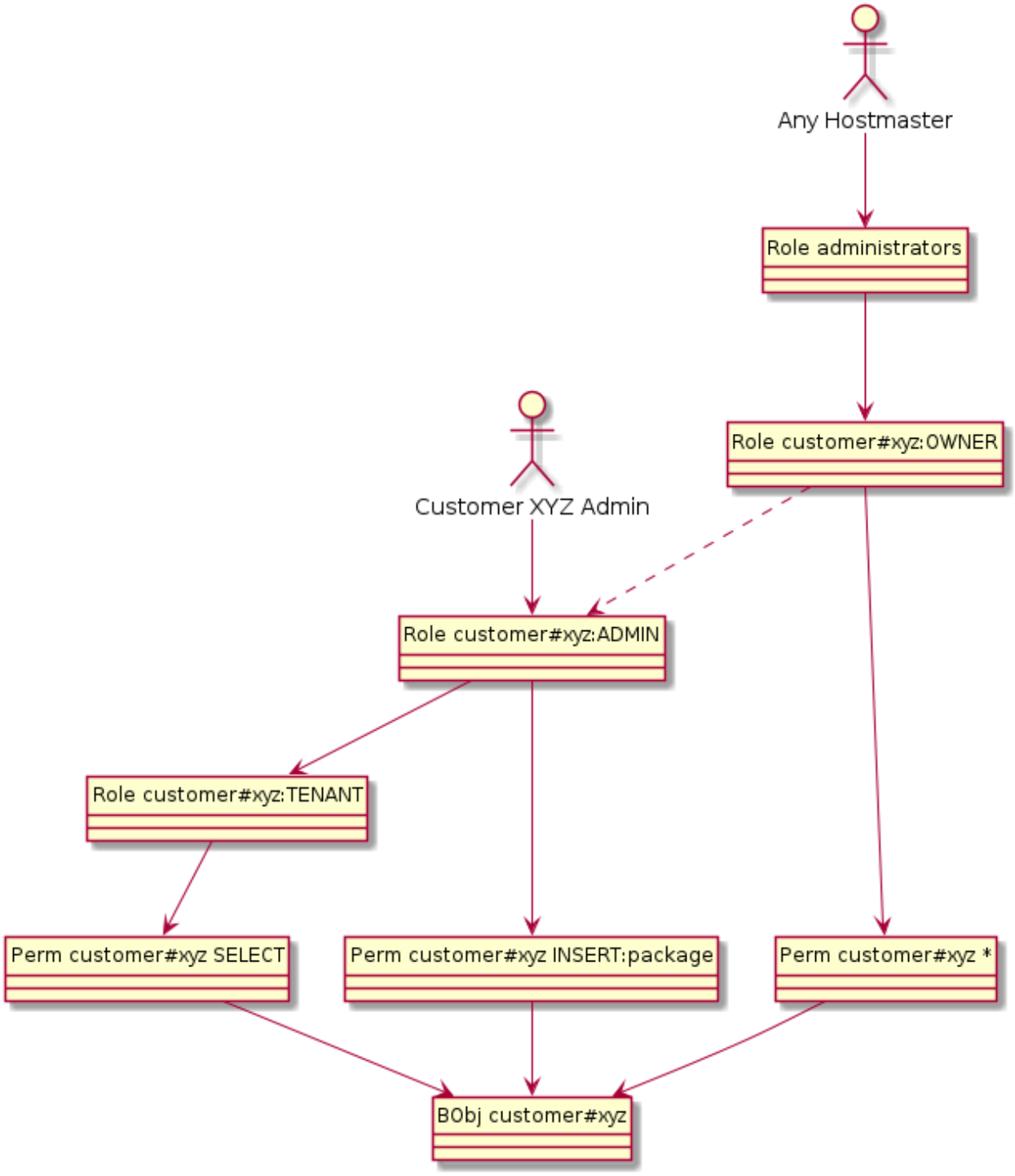


It's mostly an example hierarchy of business-object-types, but resembles a part of Hostsharing's actual hosting infrastructure.

The following diagrams show which roles are created for each business-object-type and how they relate to roles from other business-object-types.

Customer Roles

The highest level of the business-object-type-hierarchy is the *Customer*.



As you can see, there something special: From the 'Role customer#xyz:OWNER' to the 'Role customer#xyz:admin' there is a dashed line, whereas all other lines are solid lines. Solid lines means, that one role is granted to another and automatically assumed in all queries to the restricted views. The dashed line means that one role is granted to another but not automatically assumed in queries to the restricted views.

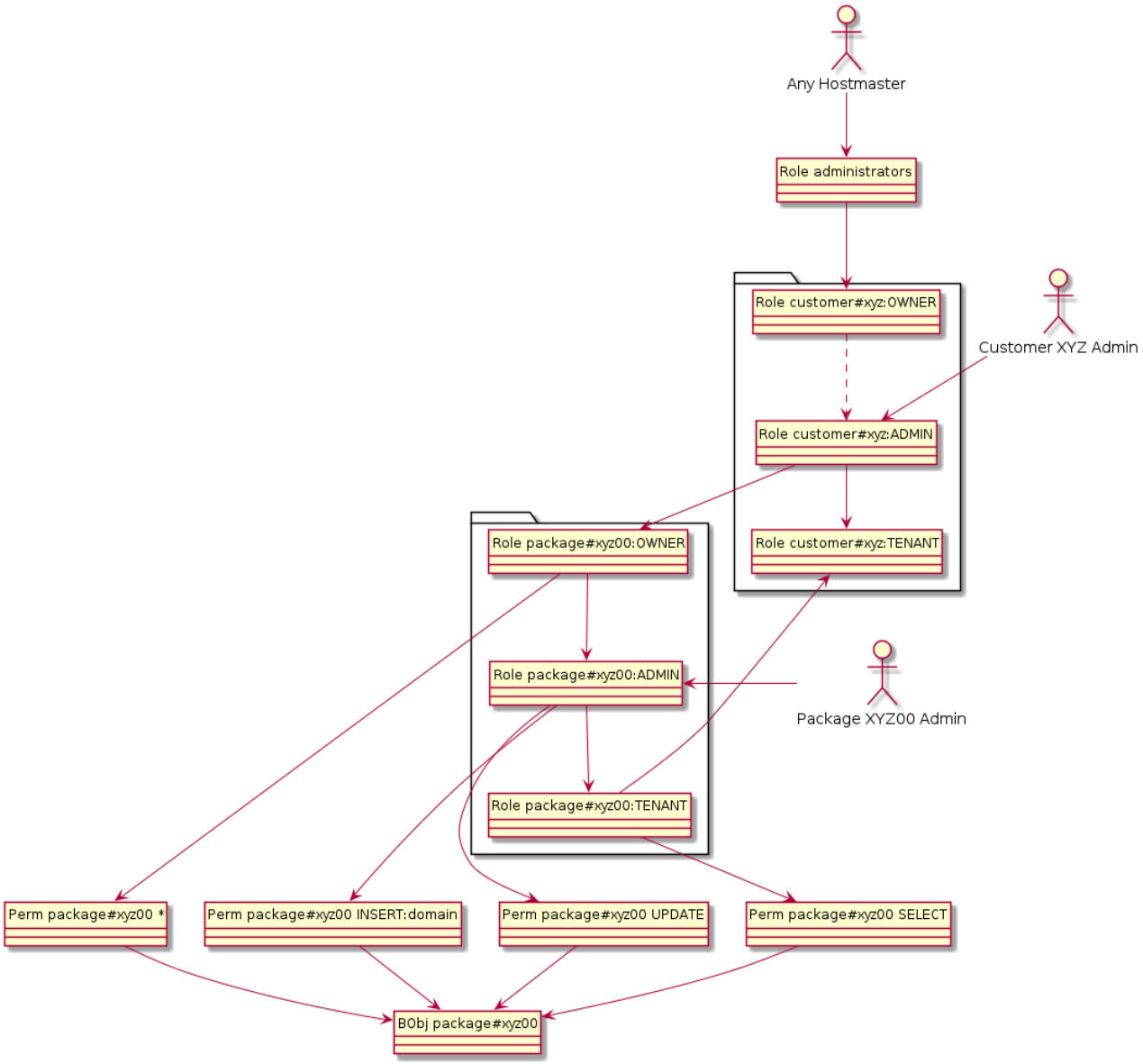
The reason here is that otherwise simply too many objects would be accessible to those with the 'administrators' role and all queries would be slowed down vastly.

Grants which are not automatically assumed are still valid grants for `hsadminng.assumedRoles`. Thus, if you want to access anything below a customer, assume its role first.

There is actually another speciality in the customer roles: For all others, a user defined by the customer gets the owner role assigned, just for the customer, the owner's role is assigned to the 'administrators' role.

Package Roles

One example of the business-object-type-level right below is the *Package*.



Initially, the customer's admin role is assigned to the package owner role. They can use the package's admin role to hand over most management functionality to a third party. The 'administrators' can get access through an assumed customer's admin role or directly by assuming the package's owner or admin role.

Performance

We did not define maximum response time in our requirements, but set a target of 7.000 customers, 15.000 packages, 150.000 Unix users, 100.000 domains and 500.000 email-addresses.

For such a dataset the response time for typical queries from a UI should be acceptable. Also, when adding data beyond these quantities, increase in response time should be roughly linear or below. For this, we increased the dataset by 14% and then by another 25%, ending up with 10.000 customers, almost 25.000 packages, over 174.000 unix users, over 120.000 domains and almost 750.000 email-addresses.

The performance test suite comprised 8 SELECT queries issued by an administrator, mostly with two assumed customer owner roles. The tests started with finding a specific customer and ended with listing all accessible email-addresses joined with their domains, unix-users, packages and customers.

Find the SQL script here: `28-hs-tests.sql`.

Two View Query Variants

We have tested two variants of the query for the restricted view, both utilizing a PostgreSQL function like this:

```
FUNCTION rbac.queryAccessibleObjectUuidsOfSubjectIds(
        requiredOp rbac.RbacOp,
        forObjectTable varchar,
        subjectIds uuid[],
        maxObjects integer = 16000)
    RETURNS SETOF uuid
```

The function returns all object uuids for which the given subjectIds (user o assumed roles) have a permission or required operation.

Let's have a look at the two view queries:

Using WHERE ... IN

```
CREATE OR REPLACE VIEW customer_rv AS
    SELECT DISTINCT target.*
      FROM customer AS target
     WHERE target.uuid IN (
        SELECT uuid
          FROM rbac.queryAccessibleObjectUuidsOfSubjectIds(
            'SELECT, 'customer', currentSubjectOrAssumedRolesUuids()));
```

This view should be automatically updatable. Where, for updates, we actually have to check for 'UPDATE' instead of 'SELECT' operation, which makes it a bit more complicated.

With the larger dataset, the test suite initially needed over 7 seconds with this view query. At this point the second variant was tried.

But after the initial query, the execution time was drastically reduced, even with different query values. Looks like the query optimizer needed some statistics to find the best path.

Using A JOIN

```
CREATE OR REPLACE VIEW customer_rv AS
    SELECT DISTINCT target.*
      FROM customer AS target
      JOIN rbac.queryAccessibleObjectUuidsOfSubjectIds(
            'SELECT, 'customer', currentSubjectOrAssumedRolesUuids()) AS allowedObjId
        ON target.uuid = allowedObjId;
```

This view cannot is not updatable automatically, but it was quite fast from the beginning.

Performance Results

The following table shows the average between the second and the third repeat of the test-suite:

Dataset	using JOIN	using WHERE IN
7000 customers	670ms	1040ms
10000 customers	1050ms	1125ms
	+43%	+57%	+8%

The JOIN-variant is still faster, but the growth in execution time exceeded the growth of the dataset.

The WHERE-IN-variant is about 50% slower on the smaller dataset, but almost keeps its performance on the larger dataset.

Both variants a viable option, depending on other needs, e.g. updatable views.

Access Control to RBAC-Objects

Access Control for business objects checked according to the assigned roles. But we decided not to create such roles and permissions for the RBAC-Objects itself. It would have overcomplicated the system and the necessary information can easily be added to the RBAC-Objects itself, mostly the `RbacGrants`.

RbacSubject

Users can self-register, thus to create a new RbacSubject entity, no login is required. But such a user has no access-rights except viewing itself.

Users can view themselves. And any user can view all other users as long as they have the same roles assigned. As an exception, users which are assigned to global roles are not visible by other users.

At least an indirect lookup of known user-names (e.g. email address of the user) is possible by users who have an empowered assignment of any role. Otherwise, it would not be possible to assign roles to new users.

RbacRole

All roles are system-defined and cannot be created or modified by any external API.

Users can view only the roles to which are granted to them.

RbacGrant

Grant can be `empowered`, this means that the grantee user can grant the granted role to other users and revoke grants to that role. (TODO: access control part not yet implemented, currently all accessible roles can be granted to other users)

Grants can be `managed`, which means they are created and deleted by system-defined rules. If a grant is not managed, it was created by an empowered user and can be deleted by empowered users.

Grants can be `assumed`, which means that they are immediately active. If a grant is not assumed, the grantee user needs to use `assumeRoles` to activate it.

Users can see only grants of roles to which they are (directly?) assigned themselves.

TODO: If a user grants an indirect role to another user, that grant would not be visible to the user. But if we make indirect grants visible, this would reveal too much information. We also cannot keep the granting user in the grant because grants must survive deleted users, e.g. if after an account was transferred to another user.