Access Roles Module

Overview

This module creates a REST API to assign new roles to identities and to query the roles already assigned on Fedora Glossary.

In roles-based access control, users or groups are not granted specific actions on resources; rather, users and groups have roles assigned to them on resources, and these roles are mapped onto permitted actions elsewhere. This makes it much easier to manage permissions globally: rarely will masses of resources need to be updated if their permissions change. Only the role-to-permission mapping will be updated. Role-based access control is a common pattern in security, providing extensible role-specific behavior while retaining straightforward management.

This module does not define any specific roles or enforce permissions granted to roles. For roles to be effective, this module must be configured alongside an authorization delegate that is aware of roles. One roles-aware authorization delegate is provided as a reference implementation, the Basic Role-based Authorization Delegate.

REST API

The module adds another REST endpoint to every Fedora Glossary path. The URL pattern is as follows:

```
<path to Container>/fcr:accessroles
```

REST methods:

method	description
GET	Retrieves the roles assigned on a resource.
GET w/effective parameter	Retrieves the effective roles assigned on a resource, which may cascade from an ancestor role assignment.
POST	Sets all the roles assigned on a resource.
DELETE	Removes any roles assigned on a resource, such that effective roles are inherited again.

The POST and GET methods currently support a JSON structure (as Content-type *application/json*) where principals are mapped to lists of roles:

```
{
  "johndoe" : [ "reader" ],
  "janedoe" : [ "writer" ],
  "freddoe" : [ "patron", "editor" ]
}
```

This module assigns one or more roles to a string, which is the name of a security principal. (java.security.Principal) The principals used in your repository environment must have unique names. You may use whatever principals you wish, but we recommend applying the appropriate standard for your environment. This module does not validate principal names.

Fedora uses a principal named "EVERYONE" to represent the general public. This principal is added to every incoming web request. You may assign any role to the EVERYONE principal.

By default, role names are not validated, since the module does not define the set of role names that may be assigned in Fedora. However, you may configure a set of specific roles and then the API will validate roles.

Example Data

```
root/  (default content roles, i.e. no roles for anyone)
 Container A  (EVERYONE => reader; johndoe => admin)
    Binary 1  (johndoe => admin)
    Container Q  (EVERYONE => reader; johndoe => admin)
        Container R (janedee => admin)
 Container B  (EVERYONE => reader; johndoe => admin)
    Container T
        Container V
 Container C
```

Inheritance of Effective Roles

Descendant resources inherit the roles assigned on ancestor resources *only if they have no roles assigned themselves*. If a resource has any roles assigned, *these assignments override ALL ancestor assignments*.

The following cases, based on the example data above, demonstrate how inheritance plays out.

1. Binary 1 of Container A only allows one principal to access the resource: *johndoe*. He will have *admin* privileges. None of the ACLs on Container A will be applied; the binary will not inherit the *EVERYONE => reader* ACL on Container A.
2. Container R, a child of Container Q, has its own content ACL: *janedee* has *admin* privileges on Container R. No one else has any access to the resource, not even the parent resource (Container Q) principals (*EVERYONE* and *johndoe*).
3. Container T, a child of Container B, has no content ACLs. So it inherits the ACLs of its most immediate ancestor with content ACLs: Container B. *EVERYONE* has reader privileges on Container T, and *johndoe* has *admin* privileges on the Container.
4. Container V also inherits the ACLs of Container B (its most immediate ancestor with content ACLs).
5. Container C has no content ACL; it inherits the ACLs of the root resource, which is to say, nothing. No one other than *fedoraAdmin* has any access to this Container.

Cascading Delete Permission

When deleting a resource, the user must have an effective role that will allow them to delete *ALL* the descendant Containers under the resource. (binaries, child Containers, etc..) If any descendant resource cannot be deleted, then the entire delete transaction will be denied.

For example, in the graph shown above, the principal *johndoe* cannot delete container A, although he has an admin role on it and its binary; that is because he does not have an effective role on Container R, the resource's grandchild, that will permit him to delete it. If he wants to delete Container A, he will first have to ask *janedee* to delete Container R.

Authorization Operations and Example

(Editor's note: this section would make more sense within the Basic Roles auth delegate documentation/page)

Order of operation:

- **Container Authentication**: A user comes into the system. They are assigned a **user principal**:
  - If they authenticate through some authentication gateway, then their principal may be generated from some of the person's attributes;
  - Whether they authenticate or not, the request will always acquire an "EVERYONE" principal.
- **Fedora Principal Provider extensions:** Principal provider extensions may bring in more principals after authentication, such as groups, from sources like LDAP.
- **Fedora Roles Authorization Delegate Queries for Assigned Roles on Content: What roles have been assigned?**
  - The authorization layer queries the requested repository resource(s) for any content-assigned roles.
  - If none are found locally, then it will query each ancestor in turn until role assignments are found.
  - If no role assignments are found in the tree of resources, then a default set of role assignments is used. (see Container C above)
- **Fedora Roles Authorization Delegate - Role Resolution: What roles does this request have?**
  - The set of principals in the request are compared to the principals in the ACLs on the resource. The roles for each matching principal in the Container ACL are the **effective roles** for the user.
  - At this point we have the effective access roles for this operation
- **Fedora Roles Authorization Delegate - Policy Enforcement: Does this role have permission to perform the requested action?**
  - Note: The Fedora Authorization Delegate is an extension point, so enforcement will vary by the chosen implementation. We assume that installations will combine the access roles module with a roles-based authorization delegate.
  - The effective roles, assigned to the user on the content, are used to determine if the user has permission to perform the action on a given resource.
  - Basic Role-based Authorization Delegate implementation does permission checks in java code:
    - Permission is determined by evaluating at a minimum the effective roles for the user on the resource in question, and the action requested.

    - In other roles-based authorization delegate implementations, more factors may also enter into the equation to determine permission.
- The authorization delegate will return a response to ModeShape, which will throw an exception to Fedora if access has been denied.
- Fedora will respond with a 403 if the given REST operation is denied.

- The one exception to this process is the **fedoraAdmin container role**. If the request has a fedoraAdmin user role (in the container), then no resource checks are made. The authorization delegate is not consulted as admins have permission to do everything. Resources will never have the fedoraAdmin role explicitly assigned to them, since it is a container role and not a content role. (e.g. a tomcat user role)

Examples:

1. Unauthenticated user requests to see Container A.
    a. The user is assigned the user principal "EVERYONE".
    b. The authorization delegate intercepts the request, gets the ACLs for Container A: "EVERYONE" => "reader" and "johndoe" => "admin".
    c. The authorization delegate compares the user principal "EVERYONE" to the principals in Container A's ACLs, and sees that "EVERYONE" matches. The effective role for this request is "reader", the role paired with the principal "EVERYONE" on the Container.
    d. The authorization delegate sees if the role "reader" can view the Container; it can.
    e. The authorization delegate returns "yes", and the request proceeds.
2. Unauthenticated user requests to see binary 1 on Container A.
    a. The user is assigned the user principal "EVERYONE".
    b. The authorization delegate intercepts the request, gets the ACLs for binary 1: "johndoe" => "admin".
    c. The authorization delegate compares the user principal "EVERYONE" to the principals in binary 1's ACLs, but does not find a match.
    d. The authorization delegate returns "no", and the request is denied.
3. Unauthenticated user requests to delete Container B.
    a. The user is assigned the user principal "EVERYONE".
    b. The authorization delegate intercepts the request, gets the ACLs for Container B: "EVERYONE" => "reader" and "johndoe" => "admin".
    c. The authorization delegate compares the principal "EVERYONE" to the principals in Container B's ACLs, and sees that "EVERYONE" matches. The effective role for this request is "reader", the role paired with the principal "EVERYONE" on the Container.
    d. The authorization delegate sees if the role "reader" can delete the Container; it cannot.
    e. The authorization delegate returns "no", and the request is denied.
4. John Doe requests to update binary 1 on Container A.
    a. The user is assigned the user principals "johndoe" and "EVERYONE".
    b. The authorization delegate intercepts the request, gets the ACLs for binary 1: "johndoe" => "admin".
    c. The authorization delegate compares the user principals "johndoe" and "EVERYONE" to the principals in binary's ACLs, and sees that "johndoe" matches. The effective role for this request is "admin", the role paired with the principal "johndoe" on the Container.
    d. The authorization delegate sees if the role "admin" can update the Container; it can.
    e. The authorization delegate returns "yes", and the request proceeds.