REST Authorization

Analysis

The optimal approach from a RESTful perspective would be to use the **OPTIONS verb** over the endpoint to discover which verbs are publicly allowed, which ones require authentication and which ones, once authenticated, are effectively available to the current user. Unfortunately, this approach could be more expensive to implement and we should avoid the trap to "duplicate" the check in both the OPTIONS than in the real verb, building a solution that rely on the @PreAuthorize security. Moreover, as the PreAuthorize security is not necessary the whole story as additional check could be performed by the dspace-api during the method execution this could lead to inconsistency or to the need to revisit our current @PreAuthorize implementations.

For these reason I suggest to don't focus on such approach at least in DSpace 7, it could be a nice addition but as it doesn't seem to be sufficient for our use case it would be not top priority (i.e. it would be beneficial for not yet identified REST client and not for our Angular UI).

An endpoint where we can "simply" check if a specific action is allowed or not, i.e. /api/authz/check?action=<:action>&resource=<:resource>, is formally not RESTful as there is no "REST resource" involved. An endpoint SHOULD represent a "collection of resource" and we should access this collection, a subset of resources or interact with a specific resource via the HTTP verbs to change its state.

Two possible approaches

1) Emulate the DSpace 6 behavior: use the resourcepolicies endpoint with an additional search method to mimic the feature of AuthorizeService.actionAnyOf method (see this internal 4Science PR) and, eventually, add special flags in the userDetails included in the JWT (i.e. an isAdmin, isCommunityAdmin, isCollectionAdmin attributes in the JWT token) - in this case it is still uncertain how to deal with the support methods provided by the org.dspace.app.util.AuthorizeUtil classes

2) Introduce the concept of "features" as an high level vision of what the user can do, where.

The structure of a feature resource could be

```
{
        id : "<eperson-uuuid>-my feature-<scope-uuid>",
        name: "my feature",
        _links: {
                scope: {
                        href: link-to-the-resource (site for "unscoped" feature
                },
                eperson: {
                        href: link-to-the-eperson-resource
                }
        }
}
```

Please note that this doesn't imply the need to keep a corresponding structure on the database as everything can be "generated on the fly" checking the resourcepolicy, the dspace configuration and our business logic.
There is no initial need to have features linked to group as at the checking time we always check permission on a specific user. In future if we want to visualize which feature are available to a specific group we can do that without touching anything assuming that features available to a group will be also "listed" as available to each member of such group.

On a such endpoint we can easily implement a **GET /api/authz/features/<:feature-id>** hardcoding the logic the feature id generation on the REST client or, better, we can add a **search method /api/authz/features/search/criteria?scope=<:dso-uuid>&eperson=<:eperson-uuid>&name=<:feature-name>**.

A null eperson can be used to identify anonymous feature, like access to the self-registration or the public statistics.

On the implementation side we could have an interface to define "feature" and a set of Beans that will be registered to provide the implementations. When a request comes we should "only" retrieve the corresponding bean and run the method to verify the permission to decide if such "feature resource" exists or not for a specific user.
If is obviously to think about future extension to retrieve all the features associate to a specific user, just iterating of the beans implementations, or provide a configuration endpoints that will list of the available features (beans).

The controversial aspect of such proposal is the definition of what is a "feature", i.e. edit of an item, withdrawn, reinstate, move are different features? self-register and user creation are different features?

Implementation

Unless otherwise specified, all DSpace v7 REST API methods/endpoints will default to ANONYMOUS access. This page details how to access restrict specific endpoints/methods by simply adding `@PreAuthorize` annotations to those methods.

The Authorizations checks in the new REST API are based on Spring Authorization annotations. Currently we only make use of the `@PreAuthorize` annotations. This means Spring will evaluate the expression within the annotations before executing the annotated method. Spring Security relies on Spring AOP Proxies to do this. This means that inner-class method calls are not evaluated by Spring security (unless you autowire a self-reference, see this StackOverflow discussion).

If the evaluation of the expression fails or returns false, Spring will not execute the method and return a 40x response. If the user is authenticated this will be a 403 Forbidden response. If the user is not authenticated, this will be a 401 Unauthorized response. This is configured using the `DSpace401AuthenticationEntryPoint class.`

The type of expressions we use are:

1. `hasPermission(#uuid, 'DSO-TYPE', 'ACTION')`: Check if the current user is allowed to execute the listed action on the specified DSpace Object (for example downloading a bitstream).

```
# Example #1:  Only allows you to access this method, if you have READ permissions on the BITSTREAM
identified by the "id" parameter
@PreAuthorize("hasPermission(#id, 'BITSTREAM', 'READ')")
public BitstreamRest findOne(Context context, UUID id) {
...
}


# Example #2: Only allows you to access this method, if you have READ permissions on the GROUP
identified by the "id" parameter
@PreAuthorize("hasPermission(#id, 'GROUP', 'READ')")
public GroupRest findOne(Context context, UUID id) {
...
}
```

2. `hasAuthority('VALUE')`: Check if the current user has a specific Spring authority. Currently there are only three authority values: `ADMIN`, `EPERSON` and `ANONYMOUS`. (for example when querying all items).

```
# Example #1: Only allows you to access this method if you are logged in as a system Administrator
@PreAuthorize("hasAuthority('ADMIN')")
public Page<BitstreamRest> findAll(Context context, Pageable pageable) {
...
}

# Example #2: Only allows you to access this method if you are currently logged in to the system
@PreAuthorize("hasAuthority('AUTHENTICATED')")
public AuthorityRest findOne(Context context, String name) {
...
}


# Example #3: Only allows you to access this method if you are anonymous (not logged in)
@PreAuthorize("hasAuthority('ANONYMOUS')")
public someMethod() {
...
}
```

For the evaluation of the `hasPermission` expressions, we wrote a custom "permission evaluator" `DSpacePermissionEvaluator` that uses a plug-in system. If one of the available plug-ins approves the requested permission, the current user is allowed to execute the action. Plugins are dynamically "discovered" using the Spring auto-wiring functionality. The plug-ins we implemented are:

1. `AuthorizeServicePermissionEvaluatorPlugin`: Check permissions based on the DSpace `AuthorizeService`. This service will validate if the authenticated user is allowed to perform an action on the given DSpace Object based on the resource policies attached to that DSpace object.
2. `EPersonRestPermissionEvaluatorPlugin`: An authenticated user is allowed to view, update or delete his or her own data. Since there are no explicit resource policies for this, the `AuthorizeService` does not cover this use case.
3. `GroupRestPermissionEvaluatorPlugin`: An authenticated user is allowed to view the information of all the groups he or she is a member of (READ permission). Since there are no explicit resource policies for this, the `AuthorizeService` does not cover this use case.
4. `AdminRestPermissionEvaluatorPlugin`: Repository Administrators are always allowed to perform any action on any DSpace object. This plugin will check if the authenticated EPerson is a repository administrator. If that is the case, the authenticated EPerson is allowed to perform the requested action.

You can easily add your own custom permission evaluator plugins by implementing the `DSpacePermissionEvaluator` interface and registering your implementation in the Spring application context.