SDN Contribution

Implementing a Read-Only Security Manager

Applies to:

Knowledge Management in NetWeaver ‘04 SPS15

SAP NetWeaver Developer Studio

Summary

A security manager is used to handle the permissions that a principal (user, group, and so on) has for individual resources. Each repository manager can have a security manager configured to instantly establish if a user has the appropriate permission to perform an operation.

When working through this tutorial, your intention may be one of the following:

- Option A: You want to implement your own repository manager and security manager from scratch.
- Option B: You want to implement only a security manager that existing repository managers can use.
- Option C: You have already worked through Thilo Brandt’s tutorial “Implementing a Repository Manager (read-only)” (RM tutorial) and you want to add your own security manager.

Depending on your intention, you should start with Option A, B, or C (see the table of contents).

Created on: May 10, 2006

Author Bio

Alexander Link has been a developer in the SAP Knowledge Management and Collaboration team since 2005. He is responsible for ACL Security.

Table of Contents

Implementing a Read-Only Security Manager 1

Applies to: 1

Summary 1

Author Bio 1

Option A: Creating a new Repository Manager Project Including a Security Manager Using the RF Component Wizard 3

Option B: Creating a Security Manager Only 3

Option C: Adding a Security Manager to an Existing Repository Manager Project 4

Coding Details 4

- getSupportedPermissions(IResourceHandle) 4
- isAllowed() 5
- Helper method: checkPermissionSettings(IPermission) and Permission config 6
- Permissions and their Effect 7

Using Caches 8

- Get cache 8
- addToCache(IResourceHandle resHandle, IPermission permission, boolean isAllowed) 8
- Boolean getFromCache(IResourceHandle resHandle, IPermission permission) 8

Security Aspects in Repository Manager Implementation 9

Configuration and Deployment 11

- Configuration files 11
- Deployment 12
- Configuration in the Portal 12

Sample Security Manager Eclipse Project 12

- SimpleRepositoryManager_security_manager.zip 12

Related Content 13

Copyright 14

Option A: Creating a new Repository Manager Project Including a Security Manager Using the RF Component Wizard

If you want to implement a repository manager and a security manager, this is the right section for you.

First you have to create a new *Portal Application Project*.
Choose *File* → *New* → *Project* → *Portal Application* → *Create new Portal Application Project*, choose a meaningful name, and confirm the wizard.

Right-click the project created and choose *New* → *Other*→ *Portal Application* → *Create a new Portal Application Object*. Then select the new project from the list, choose *RF Component Wizard 7.1.5* and choose *Next*. Now choose *Repository Manager Wizard* and choose *Next*. Specify a repository manager name, package, and a default prefix for the repository and choose *Next*.

As we develop a security manager in this tutorial, choose *Security Manager* on the *Security* tab, which causes the wizard to create an empty security manager plus its configuration. For the repository manager aspects, you should also choose *Property Manager*, *Namespace Manager*, and *Content Manager* (see the RM tutorial).

Now that we have the basic structure, you can start implementing your security manager. Continue with the *Coding Details* section and finish with the *Configuration and Deployment* section.

For more information about the implementation of the repository manager, see the RM tutorial.

Option B: Creating a Security Manager Only

Because there is no wizard for creating only a security manager, we recommend using the repository manager wizard to create a new (stand-alone) security manager. All repository manager-specific files can be deleted after creation.

First create a new repository manager project as described in option A. Note the following points:

- The repository manager name that you have to specify is the prefix of the security manager name (<YourRMName>SecurityManager.java). The security manager is created in the specified package. The repository prefix does not matter.
- Do not forget to select *Security Manager* on the *Security* tab of the Submanagers configuration.

After creating the project, you can delete all files that are not related to the security manager. Delete <*YourRMName*>*RepositoryManager.java* in *src.api* and the generated *co.xml* and *cc.xml* file in *src.config*.

Now that we have the basic structure, you can start implementing your security manager. Continue with the *Coding Details* section and finish with the *Configuration and Deployment* section.

Option C: Adding a Security Manager to an Existing Repository Manager Project

If you have already created a repository manager project with the wizard, it is unfortunately not possible to add a security manager using a wizard again. It is only possible to automatically create a security manager together with the repository manager. So you have to create the security manager manually.

First create a new class with the name <*YourName*>*SecurityManager* in an appropriate package. The class has to extend com.sap.netweaver.bc.rf.mi.**AbstractSubManager** and implement com.sap.netweaver.bc.rf.mi.security.**ISecurityManager**.

If the config class of your existing repository manager still extends **RepositoryManager**, you have to change it to **NewRepositoryManager** because the implementation uses our new manager interfaces and the repository manager would not work properly. The RM tutorial still sets the obsolete value!

Now that we have the basic structure, you can start implementing your security manager. Continue with the *Coding Details* section and finish with the *Configuration and Deployment* section.

Coding Details

To develop a security manager, you have to implement `com.sap.netweaver.bc.rf.mi.security.ISecurityManager` and you should extend `com.sap.netweaver.bc.rf.mi.AbstractSubManager`.

The `ISecurityManager` interface is pretty simple. It only requires five methods – `getSupportedPermissions(IResourceHandle)` and `isAllowed(…)` with four different signatures.

The simple security manager presented in this tutorial only supports basic read-only permissions, it is independent of the principal and it determines whether a resource is restricted based on the first character of the resource name. If the resource name starts with a '~', the resource is restricted. If it starts with any other character, it is unrestricted. For example:

/simple/test.txt	unrestricted
/simple/~test.txt	restricted

In addition, you can configure which permissions are to be allowed and which permissions are to be denied if the resource is restricted.

getSupportedPermissions(IResourceHandle)

This method should return a list of *IPermissions* that the security manager supports for the given resource. Normally, plain resources and collections are distinguished. If null is passed as a parameter, all supported permissions should be returned.

In the example security manager developed in this tutorial, only the basic read-only permissions are provided for any resource. We first initialize the list of *supportedPermissions* at startup:

```
List supportedPermissions = new ArrayList();
supportedPermissions.add(Permission.PERMISSION_LIST);
supportedPermissions.add(Permission.PERMISSION_READ_CONTENT);
supportedPermissions.add(Permission.PERMISSION_READ_NODE_PROPERTIES);
```

```
supportedPermissions.add(Permission.PERMISSION_READ_PROPERTIES);
```

The *getSupportedPermissions* method only returns that list.

isAllowed()

The *ISecurityManager* interface provides *isAllowed*() methods with four different signatures.

- The basic method `boolean isAllowed(IResourceHandle resourceHandle, IPrincipal principal, IPermission permission)` has to check whether the principal has permission for the resource.
- `boolean isAllowed(IResourceHandle resourceHandle, IPrincipal principal, List permissions)` has to check if the principal has all specified permissions for the resource handle.
- `Set isAllowed(List ridList, IPrincipal principal, IPermission permission)` returns a set containing the *IRids* of the resources for which the principal has the specified permissions.
- `Set isAllowed(List ridList, IPrincipal principal, List permissions)` does the same as the previous method, but the principal must have all specified permissions for a resource.

In the example security manager in this tutorial, the last three methods use the basic *isAllowed* method to check the permissions by calling the method once for each resource and permission.

In cases of doubt, a real security manager should always return a false result, thus denying a principal the permission! In this example, the default return value is always true for simplicity reasons.

The basic *isAllowed* method checks the first character of the resource name and determines whether permission exists for this resource or not. In addition to the RESTRICT_PREFIX ('~', which restricts resources), the system checks if permission should be granted or denied using the method `boolean checkPermissionSettings (IPermission)`.

```
public boolean isAllowed(IResourceHandle resHandle, IPrincipal principal,
IPermission permission){
      //Not cached -> get resource permission
      String name = resHandle.getRid().name().toString();
      boolean allowed = true;
      if(name != null && name.startsWith(RESTRICT_PREFIX)){
            //Resource starts with <RESTRICT_PREFIX>.
            //Get permission state from config file
            allowed = checkPermissionSettings(permission);
      } else {
            //Resource doesn't start with <RESTRICT_PREFIX>.
            //Allow everything
            allowed = true;

   }
   return allowed;
}
```

Helper method: checkPermissionSettings(IPermission) and Permission config

For demonstration purposes in the tutorial example implementation, you can configure which permissions are granted and which permissions are denied for restricted resources (resources with the '~' prefix).

The config file is a properties file with the following content. The value of each property can be modified at runtime without restarting the engine. After a timeout of one second, the properties file is read again (see implementation).

```
leaf_read_content=deny
leaf_read_properties=deny
node_list_children=deny
node_read_properties=deny
MyPermission1=allow
MyPermission2=allow
```

The path of the config file is currently hard-coded as `"C:/temp/simplesecurity.properties".` You have to copy or create the file there. When the timeout of one second has passed, the properties file is read again to identify any changes you made to the configuration (implementation in this way for testing purposes only!).
You can switch permissions between deny and allow to compare the behavior at runtime. (see "Permissions and their Effect" for more information)

If the permissions value in the config file is "deny", the method returns *false*, else it returns *true*.

```
private boolean checkPermissionSettings(IPermission permission){
      try {
            //Load Properties if last update > 1sec
            if(lastUpdate < System.currentTimeMillis() - 1000)
            {
                  temp_properties = new Properties();
                  temp_fis = new FileInputStream(permissionConfigFile);
                  temp_properties.load(temp_fis);
                  temp_fis.close();
                  lastUpdate = System.currentTimeMillis();
            }
            //Read and return whether permission is allowed or not
            return !temp_properties.getProperty(permission.getLocalName())
                        .equals("deny");
      } catch (Exception e) {
            return true;
      }

  }
```

Permissions and their Effect

The RF distinguishes the following permissions:

- For resources (leafs in the hierarchy) and collections:

  Read content (leaf_read_content):
  Permission to read the content of the resource
  deny → No access to content of resources + resource not displayed

  Read properties (leaf_read_properties):
  Permission to read the properties of the resource
  deny → No properties of resource displayed

  Write content (leaf_write_content):
  Permission to write or update the content of the resource

  Write properties (leaf_write_properties):
  Permission to write, update, or delete properties of the resource

  Delete (leaf_delete):
  Permission to delete the resource

- For collections (nodes in the hierarchy) only:

  List child nodes (node_list_children):
  Permission to retrieve the children of a collection
  deny → No access to children of collection

  Create child node (node_create_child):
  Permission to create a child in a collection

  Read node properties (node_read_properties):
  Permission to read the collection's properties or names
  deny → No properties of collection displayed

  Write node properties (node_write_properties):
  Permission to write the collection's properties or names

  Delete child node (node_delete):
  Permission to delete a child from a collection

In this tutorial additionally the following two permissions were implemented:

MyPermission1 (MyPermission1):
This permission is checked additionally to the *Read Content* permission in the ContentManager.
deny → No access to content of resources

Hint: Setting `MyPermission1=deny` and `leaf_read_content=allow` in our scenario causes resources to be listed although the user has no access to the content. When leaf_read_content is set to deny resources are not listed.

MyPermission2 (MyPermission2):
No effect

Using Caches

As security managers are heavily used, you should consider implementing caching mechanisms. The (Flexible UI) *Admin Explorer*, for example, has to call *isAllowed* multiple times for each resource (list children, read properties, and so on), for each collection renderer (tree view on the left, list view on the right) in each rendering cycle.

I implemented a basic caching mechanism in our simple security manager to give you an idea of how you could implement it.

I encapsulated all coding used for caching in an inner class, called *MyCache*. The class mainly contains a cache object (*ICache*), which represents a KM cache, the init methods, and methods to read from and write to the cache.

Get cache

```
if(repositoryManager instanceof AbstractManager){
      String cacheID = ((AbstractManager)repositoryManager).getConfig()
                        .getAttribute("securitymgr.aclcacheid", null);
      ICache cache = CacheFactory.getInstance().getCache(cacheID);
}
```

To get the cache that is configured for the security manager, we get the ID of the cache from the configuration and get the cache object from the *CacheFactory*.

addToCache(IResourceHandle resHandle, IPermission permission, boolean isAllowed)

```
String cacheEntryKey = getCacheEntryKey(resHandle, permission);
cache.addEntry(cacheEntryKey, new Boolean(isAllowed),CACHE_TTL);
```

To store information in the cache, we get the *cacheEntryKey* representing the resource-permission combination and add the *isAllowed* Boolean using this key. In addition. the CACHE_TTL (cache entry time to live. Set to 60 seconds in the example) defines how long the cache entry is valid.

Boolean getFromCache(IResourceHandle resHandle, IPermission permission)

```
String cacheEntryKey = getCacheEntryKey(resHandle, permission);
ICacheEntry entry = cache.getEntry(cacheEntryKey);
if(entry != null){
      if(!entry.isExpired()){
            return ((Boolean)entry.getObject());
      } else {
            cache.removeEntry(entry);
      }
}
return null;
```

To get an *isAllowed* state from the cache, we again create the *cacheEntryKey* and read the entry from the cache. If the entry is null, there is no result for this resource-permission combination there yet. If the entry has expired, we remove the entry from the cache and return null. If the entry is still valid, we get the Boolean stored in the entry and return it (we know it is a Boolean representing *isAllowed* for this combination).

Be aware of the caching problematic in a clustered system. Using a read/write security manager you have to ensure that cached security states are invalidated if the state is changed on another cluster

node.

Security Aspects in Repository Manager Implementation

Although a repository manager has a security manager configured, the *isAllowed*() methods of the security manager are not called automatically. To check permissions, you can either use the *isAllowed*() methods of the security manager or use the *SecurityChecker*, which dispatches the call to the configured security manager.

The *SecurityChecker* should be used to check the standard permissions. Using the *check* methods provided ensures that all necessary permissions for an action are checked, especially when complex permission checks are needed like when copying a resource from one folder to another.
To check custom permissions you have to use the *isAllowed*() methods of the security manager (which has to support the permissions).

```
//Example 1 using the SecurityChecker to check
//if the user has the Read Content Permission
try {

  this.getSecurityChecker().checkReadContent(handle);

} catch (AccessDeniedException ade) {
  throw ade;
}


//Example 2 using the SecurityManager directly to check
//if the user has the custom “MyPermission1”
ISecurityManager secMan =
  ((SimpleRepositoryManager)this.repositoryManager).getSecurityManager();

if(secMan != null){
  IUser user = AccessContextFactory.getInstance().getContext().getUser();
  IPermission permission = MyPermission.MY_PERMISSION_1;
  boolean allowed = secMan.isAllowed(handle, user, permission);
  if(!allowed) throw new AccessDeniedException(handle.getRid(),
                            permission.getName(), user.getUniqueName());
}
```

I added the following security checks to some classes of the Repository Manager Tutorial.

Manager/Method	Security Check
ContentManager.getContent(IResourceHandle)	checkReadContent
NamespaceManager.findResources(…)	checkListChildren
NamespaceManager.countResources(…)	checkListChildren
PropertyManager.getProperty/Properties(…)	checkReadProperties

The following list is copied from the RF Concepts (RCO) document and specifies generally which checks have to be performed using *SecurityChecker* for which operation (see RCO document for more information):

- `checkFind`(resource) on `findResources(`, `countResources()`:
  `LIST` (only relevant if the resource is a collection)
- `checkCreate`(resource) on `createResource()`:
  `CREATE` for resource's parent
- `checkDelete`(resource) on `deleteResource()`:
  If resource is a collection `DELETE` else `DELETE_NODE`
  and `WRITE_NODE_PROPERTIES` for resource's parent
- `checkGetContent`(resource) on `getContent()`:
  `READ_CONTENT` for the resource
- `checkSetContent`(resource) on `setContent()`:
  `WRITE_CONTENT` for the resource
- `checkGetProperty`(resource) on `getProperty()`:
  If the resource is a collection `READ_NODE_PROPERTIES` else `READ_PROPERTIES`
- `checkSetProperty`(resource) on `updateProperty()`:
  If the resource is a collection `WRITE_NODE_PROPERTIES` else `WRITE_PROPERTIES`
- `checkMove`(source, target) on `moveResource()`:
  If it is a rename (source's and target's parent are the same):
  `checkSetProperty`(parent) to check if a folder allows write
  else (if it's a real move):
  `checkDelete`(source), to check if the source can be deleted
  and if the target already exists:
  `checkDelete`(target) to check if the existing target can be overwritten
  else (if target will be newly created):
  `checkCreate`(target) to check if the non-existing target can be created
- `checkCopy`(source, destination) on `copyResource()`:
  If the source is a collection `READ_NODE_PROPERTIES` *and* `LIST`
  else `READ_PROPERTIES` *and* `READ_CONTENT`
  and if destination does not exist:
  `checkCreate`(destination), to check if destination can be created
  else (destination already exists)
  `checkSetProperty`(destination's parent), to check if folder allows write
  and if destination is not versioned
  `checkDelete`(destination) to check if the existing, non-versioned target can be overwritten
  else (if destination is versioned)
  `checkSetProperties`(destination)
  and `checkSetContent`(destination)

The remaining operations use the checks specified above:

- `getLocks()`: `checkGetProperty`()
- `lock()`: `checkSetProperty`()

- `refreshLock()`: `checkSetProperty()`
- `unlock()`: `checkSetProperty()`
- `getCollectionOrderMechanism()`: `checkGetProperty()`
- `setCollectionOrderMechanism()`: `checkSetProperty()`
- `reorderCollection()`: `checkSetProperty()`
- `getLinkDescriptor()`: `checkGetProperty()`
- `setLinkDescriptor()`: `checkSetProperty()`
- `setVersionControlEnabled()`: `checkSetProperty()`
- `checkIn()`: `checkSetProperty()` *and* `checkSetContent()`
- `updateFromVersion()`: `checkSetProperty()` *and* `checkSetContent()`

Configuration and Deployment

Configuration files

Due to problems with the generated configuration files, you must create a new configuration class (*cc.xml*) and configuration object (*co.xml*).

First create a *security_managers* folder in *src.config/install/meta/expanded/cm/repository_managers* and *src.config/install/data/cm/repository_managers*.
Then create a <*YourName*>*SecurityManager.cc.xml* file in the *meta/…/security_managers* folder and a <*YourName_lower_case*>*_ security_manager.co.xml* file.

Copy the following configuration data to the *cc.xml* and change it. Replace *YourSecurityManager* with your own manager:

```
<ConfigClass name="YourSecurityManager" extends="SecurityManagerMi"
hotReload="false" hotLoad="false" hotUnload="false">
<attribute name="class" type="class" constant="yourpackage.
YourSecurityManager"/>
<attribute name="renderingclass" type="class"
constant="com.sapportals.wcm.control.acl.DefaultPermissionControl"/>
</ConfigClass>
```

Copy the following configuration data to the *co.xml* and change it. Replace *YourSecurityManager* with your own manager:

```
<Configurable configclass="YourSecurityManager">
<property name="name" value="YourSecurityManager" />
<property name="class" value="yourpackage.YourSecurityManager" />
<property name="renderingclass"
value="com.sapportals.wcm.control.acl.DefaultPermissionControl" />
</Configurable>
```

As we have only implemented the *ISecurityManager* interface so far, we can use the *DefaultPermissionControl* to display the permissions in the *Details* dialog (*Details* → *Permissions*). The

DefaultPermissionControl lists each permission that the security manager supports and shows whether the current user has this permission or not.

The advanced *ResourcePermissionControl* lists the complete access control list (ACL). To be able to use this control, you must extend the security manager .This is not covered in this guide.

Deployment

To deploy your project, choose *File* → *Export* → *PAR File* , select the project, select *Deploy PAR* and enter or select the server. Then choose *Finish*.

You have to restart the engine now.

Configuration in the Portal

You can configure your (or any other) repository manager in the portal at *System Administration* → *System Configuration* → *Knowledge Management* → *Content Management* → *Repository Managers*.

Assign your security manager to a repository manager using the *Security Manager* property.
You can choose *ca_cm_rep_acl* as the *ACL Manager Cache*.

You have to restart the engine after configuration.

Sample Security Manager Eclipse Project

Attached to this tutorial, you can find two example security manager projects that you can import to SAP NetWeaver Developer Studio/Eclipse.

SimpleRepositoryManager_security_manager.zip

This project applies to options A and C.
The project is based on the Eclipse project provided with the RM tutorial. The project is extended with the *SimpleSecurityManager* developed in this tutorial.

SimpleSecurityManager.zip

This project applies to option B.
This project contains a stand-alone version of the *SimpleSecurityManager* developed in this tutorial.

config_and_dummy_files.zip

This archive contains the permission configuration file and some dummy resources. You have to extract the archive to the root of drive C:\.

Known Bugs

- Custom permissions are not displayed correctly in the *Details* dialog. This bug will be fixed in NW04 SPS18 and NW04s SP09.

Related Content

1. [RCO] Repository Framework Concepts, SAP AG, 2003
2. [RM Tutorial] Implementing a Repository Manager (read-only), Thilo Brandt, 2004

Copyright