Oracle® Content Services

Application Developer's Guide

10*g* Release 1 (10.1.2.2)

B25277-02

April 2006

Oracle Content Services Application Developer’s Guide, 10*g* Release 1 (10.1.2.2)

B25277-02

Primary Author: Nick Taylor

Contributing Author: Raymond Gallardo

Contributor: Marla Azriel, Simon Azriel, Vasant Kumar, Beth Morgan, Luis Saenz, Matt Shannon, Kenneth Turnbull

Contents

Preface

Audience

This document contains information on developing for Oracle Content Services Web Services.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

Printed documentation is available for sale in the Oracle Store at `http://oraclestore.oracle.com/`

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/membership/`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/documentation/`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Connecting to Oracle Content Services

The Oracle Content Services Web Services Development Kit

This guide should be used in conjunction with the Oracle Content Services Web Services Development Kit. The Development Kit can be downloaded from:

`http://www.oracle.com/technology`

Inside you will find the necessary JAR files with which to run the Web Services client. The examples in this guide are available in full within the Development Kit, along with additional documentation, the bulk tools, more samples, and the Java API Reference (Javadoc).

The following JARs from the `/lib` directory of the Development Kit must be in your `CLASSPATH` to run the examples:

- `activation`
- `axis`
- `commons-discovery-0.2`
- `commons-logging-1.0.3`
- `content-ws-client`
- `http_client`
- `jaxrpc`
- `mail`
- `saaj`
- `wsdl4j-1.5.1`
- `xmlpaserv2`

The examples in this guide are available in the Development Kit in the `sample_code/sample_webservices/src/oracle/ifs/examples/ws` directory. The Development Kit also provides documentation and a number of examples for creating, configuring, and using custom Business Process Execution Language (BPEL) workflows with Oracle Content Services. The custom workflow document provides information about how to get started with custom BPEL workflows.

The Development Kit also contains documentation for the command-line tools, which provide a way to easily load and modify a large number of Libraries or Groups.

Connecting to an Oracle Content Services Instance

The first step for a client is to authenticate with the Oracle Content Services instance. The instance is accessed at a fixed URL such as `http://servername.com/content/ws`. To connect to an instance, you will need a URL and port, as well as a valid username and password. Authentication uses the `RemoteLoginManager` class to pass the username and password to the running instance.

Note: For the example below, we will assume that your Oracle Content Services instance can accept authentication requests over an insecure connection. By default, Oracle Content Services only allows cleartext authentication over HTTPS. To change this setting, the `CleartextAuthenticationRequiresHttps` property of the Oracle Collaboration Suite domain must be set to false. See Chapter 6 of the *Oracle Content Services Administrator's Guide* for more information.

The `RemoteLoginManager` object must be retrieved from the server using its `ServiceLocator`.

```
RemoteLoginManagerServiceLocator rlmsl =
    new RemoteLoginManagerServiceLocator();
rlmsl.setMaintainSession(true);

// initialize the RemoteLoginManager
s_RLM = rlmsl.getRemoteLoginManager(new URL(serverUrl +
                                    "/RemoteLoginManager"));
```

The `RemoteLoginManager.login` method returns a set of properties about the login session (namely, the login user, session timeout, and transaction timeout). These properties are stored in an array of `NamedValue` pairs, and can be used as needed.

```
// establish a session
NamedValue[] properties = s_RLM.login(username, password, null, null);
```

After logging in, get the value of the `HEADER_COOKIE` property from the `RemoteLoginManager`. This cookie will be used to register the session with other Manager instances. In order to properly store session state, **the client must support HTTP cookies**.

```
// get the cookies
s_Cookie =
    (String) ((Stub) s_RLM)._getCall().getMessageContext().getProperty(
        HTTPConstants.HEADER_COOKIE);
```

The examples in this book use the `WsConnection` utility class to connect to an instance and retrieve Manager objects.

Note: Oracle Content Services imposes a limit on the number of concurrent sessions a user can maintain. This limit protects against Denial of Service attacks. Because HTTP is a connection-less protocol, the server has no way of knowing if a client disconnects unless explicitly alerted to by the client by way of a request (such as calling a `logout` method). Many servers, including Oracle Content Services, impose a session inactivity timeout, whereby if the user has not made a request in a certain period of time, the server frees (and disconnects) the client's inactive session. Should the client not explicitly log out, it would be possible to flood the server with new connections (and thus new user sessions) at a higher rate than the server can dispose of the inactive sessions. Thus, always invoke the RemoteLoginManager logout operation when appropriate.

Initializing Manager Classes

Manager classes must be retrieved and initialized individually before a user can perform actions with them. Each Manager is located at the base URL of the Web Services instance with /*ManagerName* appended. To retrieve an object for a Manager, you will use the corresponding Locator class.

```
DomainManagerServiceLocator dmsl = new DomainManagerServiceLocator();
dmsl.setMaintainSession(true);
```

The `setMaintainSession` method sets a flag to allow the Manager instance to be used across a persistent user session. This session must be registered with each Manager individually by passing the session cookie. Registering the session prevents having to authenticate and register with each Manager every time you want to perform an operation.

The `Manager` instance is returned from the Locator class.

```
DomainManager s_DM = dmsl.getDomainManager(new URL(serverUrl + "/DomainManager"));
```

Register the session cookie with a Manager instance by setting its `HEADER_COOKIE` property (`Stub` is a JAX-RPC class used to set properties on the Manager instance.).

```
((Stub) s_DM)._setProperty(HTTPConstants.HEADER_COOKIE, s_Cookie);
```

2 Oracle Content Services Container Manager

Oracle Content Services Web Services allow you to create Containers, which are a special type of folder used to organize Workspaces. Containers cannot contain documents, only other containers and workspaces. They have no quota and no trash.

The Container Manager is one of the simplest managers in Oracle Content Services. It allows you to create, delete, and update containers. For this example you will need to use the Container Manager to create the containers, and the Domain Manager to retrieve the root directory in which to create the containers.

2.1 Creating a Container

Once authenticated with an Oracle Content Services instance, you can create a Container using the `createContainer()` method as follows.

```
/**
 * Creates a Container in the default Domain.
 *
 * Note: The logged-in user must have the ContainerAdministrator role and the
 * logged-in session must be in domain-administration mode.
 */
public static Item createContainer(String containerName, String description)
    throws FdkException, RemoteException
{
    // get the Manager instances we need
    ContainerManager cm = WsConnection.getContainerManager();
    DomainManager dm = WsConnection.getDomainManager();

    // get default domain
    Item defaultDomain = dm.getDefaultDomain(null);

    // create the Container definition
    NamedValue[] cnDef = WsUtility.newNamedValueArray(new Object[][] {
        { Attributes.NAME, containerName },
        { Attributes.DESCRIPTION, description } });

    // container attribute request
    AttributeRequest[] cont_attr = new AttributeRequest[] {
        WsUtility.newAttributeRequest(Attributes.PATH),
        WsUtility.newAttributeRequest(Attributes.DESCRIPTION) };

    // create the Container
    Item container = cm.createContainer(domain.getId(), cnDef, cont_attr);

    WsUtility.log("container attributes");
    WsUtility.log(WsUtility.INDENT, container);
    return container;
}
```

First, the `WsConnection` class is used to retrieve the Container and Domain Managers.

The Domain Manager is used to retrieve a domain, which is a root folder in Oracle Content Services. Typically, there will be only one domain, unless there are multiple sites being hosted in a single instance. This example creates a Container in the default domain (the root, in this case), which is retrieved with the `getDefaultDomain` call.

To create a container the logged-in user must be in *domain administration mode*. This mode is used when you want to perform high-level actions such as adding, deleting, or modifying a Container. This mode is used throughout Oracle Content Services, and allows different operations on each Manager. In the case of Containers, all operations must be made in administrator mode. Switching to this mode will be shown later in this chapter.

Oracle Content Services returns the domain object in the form of an `Item`, which is a type representing any kind of persistent repository object, such as a Document, Folder, User, Group, or Category. An `Item` always has 3 default attributes:

1. `id` (long) - the ID of the object in the repository.
2. `name` (String) - the name of the item as stored in the repository. This value is null if the item does not have a name.
3. `type` (String) - the type of the object. All available type constants are defined in `oracle.ifs.fdk.ItemTypes`.

When creating a new container using the `ContainerManager.createContainer()` method, you must pass in a set of attributes describing the properties of the container, such as the name and description. This set is passed in an array of `NamedValue` objects.

The `cont_attr` parameter has special importance. An item may also contain a set of optional attributes that are filled in when the item object is created only if the caller requests them. These are called *attribute requests*. An attribute request is performed by creating an `AttributeRequest` array, and populating it with a list of attributes. Using attribute requests, you can fetch additional attributes about the item by passing (possibly nested) arrays of attribute requests. This allows for retrieval of entire trees of related objects and their attributes in one call. When the call containing the attribute request is made (in this case the `createContainer` call), it returns the requested attributes into an `Item` object (in this case `container`). The attributes in the item can then be retrieved using the Item method `getRequestedAttributes()`. Attribute requests are used throughout Oracle Content Services, and are discussed in more detail in **Chapter 7**.

In the last line before returning, the returned attributes are passed to a log method belonging to the `WsUtility` class, which writes a log entry for the action that took place. Inside the log method, the attributes are retrieved using the `getRequestedAttributes` method, which returns an array of `NamedValue` pairs (each containing an `Attribute` and a value).

```
public static void log(String indent, Item item)
{
    if (item == null)
    {
      log(indent, "Item is null ");
      return;
```

```
        }
        log(indent, "Item [name = " + item.getName() + "]");

        NamedValue[] attributes = item.getRequestedAttributes();
        int len = (attributes == null) ? 0 : attributes.length;

        if (len > 0)
        {
            log(indent, "Requested Attributes");
        }
        for (int i = 0; i < len; i++)
        {
            log(indent + INDENT, attributes[i]);
        }
    }
```

2.2 Deleting a Container

To delete a container, simply call the `deleteContainer()` method on the `ContainerManager`, passing in the Container ID as an argument. Note that this example assumes the session is still in administrator mode.

```
/**
 * Deletes the specified Container.
 *
 * Note: The logged-in user must have the ContainerAdministrator role and the
 * logged-in session must be in domain administration mode.
 */
public static void deleteContainer(Item container)
    throws FdkException, RemoteException
{
    // delete Container
    ContainerManager cm = WsConnection.getContainerManager();
    cm.deleteContainer(container.getId(), null);
}
```

2.3 Running the Code

Before calling these methods, remember that you must be in domain administration mode. To switch to this mode, connect to the Session Manager and call the `setSessionMode()` method as shown below.

Be sure to log out from the Web Services instance at the end using the `WsConnection.logout()` method.

```
private static WsConnection s_WsCon;

public static void main(String[] args)
{
    try
    {
        try
        {
            // get property object
            Properties prop = WsUtility.getProperty(args[0]);

            // URL to content services web services servlet
            String serverUrl = "http://" + prop.getProperty("hostname") + ":"
                + prop.getProperty("port") + "/content/ws";
                // authenticate to content services
                s_WsCon = WsConnection.login(serverUrl, prop.getProperty("user"),
                    prop.getProperty("password"));

                // create Container
                Item newContainer = createContainer("MyContainer", "This is a example
                    container");

                // delete Container
                deleteContainer(newContainer);
            }
            finally
            {
                s_WsCon.logout();
            }
        }
        catch (Throwable t)
        {
            t.printStackTrace();
        }
    }
```

3

Oracle Content Services Workspaces

Oracle Content Services Web Services allows creation of Workspaces, which are special folders that store content, have a trash folder, and can have an associated quota.

For this example you will need (in addition to the Workspace Manager):

- File Manager - For resolving path names
- Security Manager - For setting the security policy of a Workspace
- Session Manager - To access administrator mode
- User Manager - To fetch user IDs for setting permissions on the manager

Also, you will need a container in which to create the workspace. This must be a container on which the user has LibraryAdministrator access. You can also create Workspaces directly inside a Domain.

Note: The "Workspaces" referred to in this chapter are called "Libraries" in the rest of Oracle Content Services.

Creating a Workspace

A Workspace must be created inside a container or domain. You can retrieve the Container by resolving its path with the `resolvePath` method of the `FileManager`.

```
/**
  * Creates a Workspace in the specified Container and assigns the
  * WorkspaceAdministrator, WorkspaceManager, and WorkspaceAuthor Roles
  * to the specified users.
  *
  * Note: The logged-in user calling this method must have the
  *       LibraryAdministrator or WorkspaceCreator Role and the
  *       logged-in session must be in domain-administration mode.
  */
public static Item createWorkspace(
    String containerPath,
    String workspaceName,
    String workspaceDesc,
    String admin,
    String author,
    String reader) throws FdkException, RemoteException
{
    // get the Manager instances
    UserManager um = s_WsCon.getUserManager();
    FileManager fm = s_WsCon.getFileManager();
    SecurityManager scm = s_WsCon.getSecurityManager();
```

```
WorkspaceManager wm = s_WsCon.getWorkspaceManager();

// get the user Items
Item adminUser = um.getUser(admin, null);
Item authorUser = um.getUser(author, null);
Item readerUser = um.getUser(reader, null);

// get the Container Item from its path
Item container = fm.resolvePath(containerPath, null);
```

The roles for each workspace are specified by a *security configuration*, which is composed of a set of *grants*. A grant consists of a grantee user and a set of roles for that user. Each grant is represented by two name/value pairs (stored in `NamedValue` objects). The first pair associates the `GRANTEE` Attribute with the user ID for the grantee user. The second pair associates the `ROLES` Attribute with the set of role IDs being granted to that user (adminstrator, reader, or author). Note that more than one role can be granted to each user, so the `ROLES` Attribute can be associated with an array of roles.

In this example there are three users being granted access to the workspace. The administrator role is the least restricted, and allows all workspace actions, such as creating and deleting, version control, and others. The reader can view items in the workspace but not change them. The author can do all the tasks associated with adding and deleting content.

After creating each of the grant definitions, create the security configuration by forming a set from the grants. This set is stored as an array of `NamedValueSet` objects.

Note that when creating the grant definitions, this example uses the `newNamedValueArray` and the `newNamedValueSet` methods of the `WsUtility` class. These methods are simple wrappers for object instantiation.

```
// get the user Items
Item adminUser = um.getUser(admin, null);
Item authorUser = um.getUser(author, null);
Item readerUser = um.getUser(reader, null);

// get the Roles to grant for the Workspace's SecurityConfiguration
Item adminRole = scm.getRoleByName("ADMINISTRATOR", null);
Item authorRole = scm.getRoleByName("AUTHOR", null);
Item readerRole = scm.getRoleByName("READER", null);

// create a Grant definition for the Workspace administrator
//    -> notice that a Grant definition is made up of a
//       GRANTEE and a set of ROLES (here we use just one Role)
NamedValue[] wsAdminGrant = WsUtility.newNamedValueArray(new Object[][] {
    { Attributes.GRANTEE, new Long( adminUser.getId()) },
    { Attributes.ROLES, new long[] { adminRole.getId() } } });

// create a Grant definition for the Workspace author
NamedValue[] wsAuthorGrant = WsUtility.newNamedValueArray(new Object[][] {
    { Attributes.GRANTEE, new Long( authorUser.getId()) },
    { Attributes.ROLES, new long[] { authorRole.getId() } } });

// create a Grant definition for the Workspace reader
NamedValue[] wsReaderGrant = WsUtility.newNamedValueArray(new Object[][] {
    { Attributes.GRANTEE, new Long( readerUser.getId()) },
    { Attributes.ROLES, new long[] { readerRole.getId() } } });
    // create the Grant array definition
    //   -> notice that the Grant array definition is made up of three
    //      NamedValue arrays, one for each Grant definition
    //   -> each Grant definition (NamedValue[]) is wrapped in a
    //      NamedValueSet instance to avoid the use of two-dimensional arrays
    NamedValueSet[] wsGrants = new NamedValueSet[] {
        WsUtility.newNamedValueSet(wsAdminGrant),
        WsUtility.newNamedValueSet(wsAuthorGrant),
        WsUtility.newNamedValueSet(wsReaderGrant)
    };
```

Once the security configuration is set up, pass it as the parameter for the `Attributes.SECURITY_CONFIGURATION` constant, along with the Name and Description for the workspace, as shown below. Then, create the workspace using the `createWorkspace()` method of the `WorkspaceManager`, as shown.

```
    // create the Workspace definition
    //   -> notice that the Grant array definition is a nested definition
    //      passed in as the value of the SECURITY_CONFIGURATION NamedValue
    NamedValue[] wsDef = WsUtility.newNamedValueArray(new Object[][] {
        { Attributes.NAME, workspaceName },
        { Attributes.DESCRIPTION, workspaceDesc },
        { Attributes.SECURITY_CONFIGURATION,
            WsUtility.newNamedValueArray(new Object[][] {
            { Attributes.GRANTS, wsGrants } }) } });

    // workspace attribute request
    AttributeRequest[] workspace_attr =
        WsUtility.newAttributeRequestArray(new String[] {
        Attributes.DESCRIPTION,
        Attributes.TRASH_FOLDER,
        Attributes.JOINABLE,
        Attributes.JOINABLE_WORKSPACE_DESCRIPTION });

    // create the Workspace using the Workspace definition
    //    -> no workflow parameters are required (second argument is null)
    Item workspace = wm.createWorkspace(container.getId(), null, wsDef, workspace_
                    attr);

    WsUtility.log("workspace attributes");
    WsUtility.log(WsUtility.INDENT, workspace);

    return workspace;
}
```

Deleting a Workspace

To delete a workspace, fetch the Workspace ID and call the `deleteWorkspace()` method on the `WorkspaceManager`.

```
/**
 * Deletes the specified Workspace.
 *
 * Note: The logged-in user calling this method must have the
 *       LibraryAdministrator or WorkspaceCreator Role and the
 *       logged-in session must be in domain-administration mode.
 */
public static void deleteWorkspace(Item workspace)
```

```
    throws FdkException, RemoteException
{
    // delete the Workspace
    //   -> no definition is required
    WorkspaceManager wm = WsConnection.getWorkspaceManager();
    wm.deleteWorkspace(workspace.getId(), null);
}
```

Running the Code

To run the code, connect to an Oracle Content Services instance, and call the create and delete methods on a Workspace. Remember to switch to domain administrator mode before performing any operations.

```
private static WsConnection s_WsCon;

public static void main(String[] args)
{
    try
    {
        try
        {
           // get property object
           Properties prop = WsUtility.getProperty(args[0]);

           // authenticate to Oracle Content Services
           String serverUrl = "http://" + prop.getProperty("hostname") + ":"
               + prop.getProperty("port") + "/content/ws";

           s_WsCon = WsConnection.login(serverUrl, prop.getProperty("user"),
                     prop.getProperty("password"));
            // URL to Oracle Content Services web services servlet
            String serverUrl = "http://yourserver.com:7777/content/ws";

            // authenticate to content services
            s_WsCon = WsConnection.login(serverUrl, "jon", "welcome1");

            // switch to admin mode
            SessionManager sm = s_WsCon.getSessionManager();
            sm.setSessionMode(FdkConstants.SESSION_MODE_DOMAIN_ADMINISTRATION,
                null);

            // create a Workspace
            Item newWorkspace = createWorkspace("/oracle/ifs/dev", "MyWorkspace",
                "this is an example Workspace", "ray", "tanya", "ellie");

            // delete the Workspace
            deleteWorkspace(newWorkspace);
        }
        finally
        {
            s_WsCon.logout();
        }
    }
    catch (Throwable t)
    {
        t.printStackTrace();
    }
}
```

4 Oracle Content Services Group Management

Oracle Content Services Web Services offers the ability to organize users into *groups* using the `GroupManager` class. Groups allow user data to be manipulated and passed around all at once, instead of by selecting individual users. This class allows you to create new groups and add or remove users to and from those groups. This chapter illustrates basic group management using the `GroupManager` class, with an example that performs simple group operations.

4.1 Creating a Group

To create a group, you will need a GroupManager (for group operations) and a UserManager (to retrieve user IDs for adding to the group). The process is to generate member lists for the group, create a group definition based on these member lists (and the other group parameters), and then create the group based on the group definition.

There are two classes of users for a group: **managers** and **members**. A manager is allowed to perform administrative actions on a group, such as adding and deleting users, and changing the group properties. A member simply belongs to the group and cannot edit the group at all.

This example takes a member list and a manager list as parameters, which are reconstructed as `Item` arrays and passed to the group definition under the attributes `MEMBER_LIST` and `MANAGER_LIST`.

```
/**
 * Create a group with specified members and managers
 */
public static Item createGroup(String groupName, String groupDesc,
    String[] members, String[] managers) throws RemoteException, FdkException
{
    // get the Manager instances we need
    GroupManager gm = WsConnection.getGroupManager();
    UserManager um = WsConnection.getUserManager();

    // get members list length
    int len = (members != null) ? members.length : 0;

    // initalize member list
    long mbrList[] = new long[len];

    for (int i = 0; i < len; i++)
    {
        // get member
        Item member = um.getUser(members[i], null);
        // get member ID
        mbrList[i] = member.getId();
    }

    // get managers list length
    len = (managers != null) ? managers.length : 0;
    // initalize manager list
    long mgrList[] = new long[len];
    for (int i = 0; i < len; i++)
    {
        // get manager
        Item manager = um.getUser(managers[i], null);
        // get manager ID
        mgrList[i] = manager.getId();
    }
```

For this example, the attributes are two arrays of user IDs, one for the managers you would like to assign to the group (in this example, `mgrList`), and another for the members (`mbrList`), as well as the name and description of the group.

```
    // create group definition
    NamedValue[] createGroupDef =
        WsUtility.newNamedValueArray(new Object[][] {
            { Attributes.NAME, groupName },
            { Attributes.DESCRIPTION, groupDesc },
            { Attributes.MANAGER_LIST, mgrList },
            { Attributes.MEMBER_LIST, mbrList } });

    // group attribute request
    AttributeRequest[] group_attr =
    WsUtility.newAttributeRequestArray(new String[] {
        Attributes.DESCRIPTION,
        Attributes.MEMBER_LIST, Attributes.GROUP_MEMBER_LIST,
        Attributes.MANAGER_LIST });

    // create group
    //     -> no AttributeRequest is specified (second argument is null)
    Item gp = gm.createGroup(createGroupDef, group_attr);

    // log group info
    WsUtility.log(WsUtility.INDENT, gp);

    return gp;
}
```

4.2 Adding and Removing Members

In the previous method, you added both a member and a manager when creating the group. Suppose you want to add more members. The following code demonstrates how to use the `GroupManager.addUsers()` method to add a member or list of members to an existing group.

```
/**
 * Adds members to an existing Group.
 *
 *
 */
public static Item addMember(Item group, String[] members)
    throws FdkException, RemoteException
```

```
{
    // get the Manager instances we need
    GroupManager gm = s_WsCon.getGroupManager();
    UserManager um = s_WsCon.getUserManager();

    // get members list length
    int len = (members != null) ? members.length : 0;
    // initalize member list
    long mbrList[] = new long[len];
    for (int i = 0; i < len; i++)
    {
        // get member
        Item member = um.getUser(members[i], null);
        // get member id
        mbrList[i] = member.getId();
    }

    // add members definition
    NamedValue[] addMbrDef = WsUtility.newNamedValueArray(new Object[][] { {
        Attributes.MEMBER_LIST, mbrList } });

    // group attributes request
    AttributeRequest[] group_attr =
        WsUtility.newAttributeRequestArray(new String[] {
        Attributes.MEMBER_LIST });

    // add group member
    Item gp = gm.addUsers(group.getId(), addMbrDef, group_attr);

    // log group info
    WsUtility.log(WsUtility.INDENT);
    WsUtility.log("Added members");
    WsUtility.log(WsUtility.INDENT, gp);

    return gp;
}
```

Just as in the previous example, the first step is to generate a members list by fetching each user (as `Item` objects) using a call to the `UserManager`. Then, generate an array of `NamedValue` pairs from the member list, pairing each entry with the `Attributes.MEMBER_LIST` attribute with each member you want to add. Also, create an `AttributeRequest` to retrieve the member list (using the `MEMBER_LIST` attribute). Finally, pass the member list and the attribute request to the `addUsers` method in the `GroupManager`.

The process for removing members from a group is similar, as shown below. Note that while we do not need to use an attribute request here, we do so in order to log the attributes of the deleted item.

```
/**
 * Removes members from an existing Group.
 */
public static Item removeMember(Item group, String[] members)
    throws FdkException, RemoteException
{
    // get the Manager instances we need
    GroupManager gm = s_WsCon.getGroupManager();
    UserManager um = s_WsCon.getUserManager();

    // get members list length
    int len = (members != null) ? members.length : 0;

    // initalize members list
    long mbrList[] = new long[len];
    for (int i = 0; i < len; i++)
    {
        // get member
        Item member = um.getUser(members[i], null);

        // get member id
        mbrList[i] = member.getId();
    }

    // remove member definition
    NamedValue[] removeMbrDef = WsUtility.newNamedValueArray(new Object[][] { {
        Attributes.MEMBER_LIST, mbrList } });

    // group attributes request
    AttributeRequest[] group_attr =
        WsUtility.newAttributeRequestArray(new String[] {
        Attributes.MEMBER_LIST });

    // remove group member
    Item gp =  gm.removeUsers(group.getId(), removeMbrDef, group_attr);

    // log group info
    WsUtility.log("Removed members");
    WsUtility.log(WsUtility.INDENT, gp);

    return gp;
}
```

To add and delete managers instead of members, the process is the same. Simply get a manager user, associate it with an `Attributes.MANAGER_LIST` attribute in a `NamedValue` pair, and pass it in an array to `addUsers()`.

```
/**
   * Adds managers to an existing Group.
   */
public static Item addManagers(Item group, String[] managers)
    throws RemoteException, FdkException
{
    // get the Manager instances we need
    GroupManager gm = s_WsCon.getGroupManager();
    UserManager um = s_WsCon.getUserManager();

    // get managers list length
    int len = (managers != null) ? managers.length : 0;

    // initalize managers list
    long mgrList[] = new long[len];
    for (int i = 0; i < len; i++)
    {
        // get manager
        Item manager = um.getUser(managers[i], null);

        // get manager id
        mgrList[i] = manager.getId();
    }
```

```
        // add manager definition
        NamedValue[] addMgrDef = WsUtility.newNamedValueArray(new Object[][] { {
            Attributes.MANAGER_LIST, mgrList } });

        // group attributes request
        AttributeRequest[] group_attr =
            WsUtility.newAttributeRequestArray(new String[] {
            Attributes.MANAGER_LIST });

        // add group manager
        Item gp = gm.addUsers(group.getId(), addMgrDef, group_attr);

        // log group info
        WsUtility.log("Adds managers");
        WsUtility.log(WsUtility.INDENT, gp);

        return gp;
    }


    /**
     * Removes managers from an existing Group.
     */
    public static Item removeManagers(Item group, String[] managers)
        throws FdkException, RemoteException
    {
        // get the Manager instances
        GroupManager gm = s_WsCon.getGroupManager();
        UserManager um = s_WsCon.getUserManager();

        // Get managers list length
        int len = (managers != null) ? managers.length : 0;

        // initalize manager list
        long mgrList[] = new long[len];

        for (int i = 0; i < len; i++)
        {
            // get manager
            Item manager = um.getUser(managers[i], null);

            // get manager id
            mgrList[i] = manager.getId();
        }

        // remove manager definition
        NamedValue[] removeMgrDef = WsUtility.newNamedValueArray(new Object[][] { {
            Attributes.MANAGER_LIST, mgrList } });

        // group attributes request
        AttributeRequest[] group_attr =
            WsUtility.newAttributeRequestArray(new String[] {
            Attributes.MANAGER_LIST });

        // remove group manager
        Item gp = gm.removeUsers(group.getId(), removeMgrDef, group_attr);

        // log group info
        WsUtility.log("Removed managers");
        WsUtility.log(WsUtility.INDENT, gp);
    return gp;
}
```

4.3 Deleting a Group

Deleting a group is as simple as calling a single method, `deleteGroup()`. Groups must be deleted by group ID.

```
/**
 * Delete group
 */
public static void deleteGroup(Item group)
    throws FdkException, RemoteException
{
    // get the Manager instance
    GroupManager gm = WsConnection.getGroupManager();

    // delete group
    gm.deleteGroup(group.getId());
}
```

4.4 Running the Code

To run the code, connect to a running Oracle Content Services instance, create a group, and start performing group operations. Be sure to log out from the Web Services instance at the end using the `WsConnection.logout()` method.

```
private static WsConnection s_WsCon;

public static void main(String[] args)
{
    try
    {
        Properties prop = WsUtility.getProperty(args[0]);

        // URL to content services web services servlet
        String serverUrl = "http://" + prop.getProperty("hostname") + ":"
                + prop.getProperty("port") + "/content/ws";

        // authenticate to content services
        s_WsCon = WsConnection.login(serverUrl, prop.getProperty("user"),
                prop.getProperty("password"));

        try
        {
            // create group
            Item newGroup = createGroup(
                            "myGroup2",
                            "This is a example group",
                            new String[] { prop.getProperty("member") },
                            new String[] { prop.getProperty("manager"),
                                           prop.getProperty("user")}
                            );

            // remove member
            newGroup = removeMember(newGroup, new String[] {
                prop.getProperty("member") });

                // add member
                newGroup = addMember(newGroup, new String[] {
                               prop.getProperty("member") });

                // remove manager
                newGroup = removeManagers(newGroup, new String[] {
                               prop.getProperty("manager") });

                // add manager
                newGroup = addManagers(newGroup, new String[] {
                               prop.getProperty("manager") });

                // delete group
                deleteGroup(newGroup);
            }
            finally
            {
                s_WsCon.logout();
            }
        }
        catch (Throwable t)
        {
            t.printStackTrace();
        }
    }
```

5 Oracle Content Services Document Operations

Oracle Content Services provides a set of Managers for document creation, deletion, and manipulation. Documents are created within workspaces, and can be copied, moved, deleted, or moved to the trash.

For these examples, you will need:

- FileManager - for creating files and folders
- TrashManager - for deleting documents
- CommonManager - for retrieving common Oracle Content Services items, namely the Trash

Creating Folders and Documents

To create a folder, you will need a folder name, a description, and the ID of a parent folder. Pass the ID to the FileManager, along with the folder definition, and call the `createFolder()` method.

```
/**
* Creates a folder in the specified destination folder.
*/
public static Item createFolder(String folderName, String folderDesc,
    Item parent) throws RemoteException, FdkException
{
    FileManager fm = WsConnection.getFileManager();

    // create folder definition
    NamedValue[] folderDef =
        WsUtility.newNamedValueArray(new Object[][] {
            { Attributes.NAME, folderName },
            { Attributes.DESCRIPTION, folderDesc } });

    Item folder = fm.createFolder(parent.getId(), folderDef, null);

    // log the created folder
    WsUtility.log(WsUtility.INDENT, folder);

    return folder;
}
```

To create a document, the process is nearly identical, except you should call the `createDocument()` method. Also, instead of passing the parent folder ID directly to the method, you should include it as the value for the `Options.DESTFOLDER` constant in the document definition.

```
/**
 * Creates a document.
 */
public static Item createDocument(String docName, String docDesc, Item folder)
    throws RemoteException, FdkException
{
    FileManager fm = WsConnection.getFileManager();

    // document attribute request
    AttributeRequest[] doc_attr = new AttributeRequest[] {
        WsUtility.newAttributeRequest(Attributes.PATH),
        WsUtility.newAttributeRequest(Attributes.DESCRIPTION),
        WsUtility.newAttributeRequest(Attributes.VERSIONS) };

    // create document
    Item document = fm.createDocument(
    WsUtility.newNamedValueArray(new Object[][] {
        { Attributes.NAME, docName },
        { Attributes.DESCRIPTION, docDesc },
        { Options.DESTFOLDER, new Long(folder.getId()) } }), null, doc_attr);

    WsUtility.log("document attributes");
    WsUtility.log(WsUtility.INDENT, document);

    return document;
}
```

Copying or Moving a Document

To copy a document to another folder, create a "copy definition" by putting the `DESTFOLDER` option and its value into a `NamedValueArray`. Then, use the FileManager's `copy()` method.

```
/**
 * Copies a Document to a destination folder.
 */
public static void copyDocument(Item document, Item destFolder)
    throws RemoteException, FdkException
{
    // get the Manager instances
    FileManager fm = s_WsCon.getFileManager();
    CommonManager cm = s_WsCon.getCommonManager();

    // copy document
    NamedValue[] copyDef = WsUtility.newNamedValueArray(new Object[][] { {
        Options.DESTFOLDER, new Long(destFolder.getId()) } });

    NamedValueSet[] copyDefs = new NamedValueSet[] {
        WsUtility.newNamedValueSet(copyDef) };

    Item[] doc = fm.copy(new long[] { document.getId() }, null, copyDefs);

    // document attribute request
    AttributeRequest[] doc_attr = new AttributeRequest[] {
        WsUtility.newAttributeRequest(Attributes.PATH) };

    WsUtility.log("copy document attributes");
    WsUtility.log(WsUtility.INDENT, cm.getItem(doc[0].getId(), doc_attr));
}
```

To move a document the process is identical, except you will need to use the `move()` method.

```
/**
 * Moves a Document to a destination folder.
 */
public static void moveDocument(Item document, Item destFolder)
    throws RemoteException, FdkException
{
    // get the Manager instances
    FileManager fm = s_WsCon.getFileManager();
    CommonManager cm = s_WsCon.getCommonManager();

    NamedValue[] moveDef = WsUtility.newNamedValueArray(new Object[][] { {
        Options.DESTFOLDER, new Long(destFolder.getId()) } });

    NamedValueSet[] moveDefs = new NamedValueSet[] {
        WsUtility.newNamedValueSet(moveDef) };

    Item[] doc = fm.move(new long[] { document.getId() }, null, moveDefs);

    // document attribute request
    AttributeRequest[] doc_attr = new AttributeRequest[] {
        WsUtility.newAttributeRequest(Attributes.PATH) };

    WsUtility.log("move document attributes");
    WsUtility.log(WsUtility.INDENT, cm.getItem(doc[0].getId(), doc_attr));
}
```

Deleting Documents and Folders

Deleting documents and folders involves moving them to the *trash*, and then emptying the trash. The trash is simply a folder with a special designation as the `TRASH_FOLDER`, so moving a document or folder only requires the use of the `FileManager`. The trash belongs to the Library containing the folder or document. In order to empty the trash and thus delete the folders or documents permanently, you will need to use the `TrashManager`. To empty the trash, call the `emptyTrash()` method, and pass in the ID of the trash folder.

This example also uses the `CommonManager`, which allows you to retrieve any kind of item, provided the logged-in user has permission, using the `getItem()` method. Using the `CommonManager`, you can retrieve the `Item` corresponding to the document to delete. The `getItem()` call is also combined with an `AttributeRequest` to retrieve the `Item` corresponding to the trash folder.

```
/**
 * Deletes a Document and empties the Trash.
 */
public static void deleteDocument(Item document)
    throws RemoteException, FdkException
{
    // get the Manager instances
    CommonManager cm = s_WsCon.getCommonManager();
    FileManager fm = s_WsCon.getFileManager();
    TrashManager tm = s_WsCon.getTrashManager();
    // TRASH_FOLDER AttributeRequest array
    AttributeRequest[] trash_attr =
        WsUtility.newAttributeRequestArray(Attributes.TRASH_FOLDER);

    // get TRASH_FOLDER attribute
    document = cm.getItem(document.getId(), trash_attr);
    NamedValue[] attrs = document.getRequestedAttributes();
    Map attrMap = ClientUtils.namedValuesToMap(attrs);
    Item trashItem = (Item) attrMap.get(Attributes.TRASH_FOLDER);

    // delete the Document and empty the Trash
    fm.delete(new long[] { document.getId() }, null, null);
    tm.emptyTrash(trashItem.getId());
}
```

Running the Code

To run the code, connect to a Oracle Content Services instance and call the document and folder methods. Remember to log out of the Web Services instance using the `logout()` method of the `WsConnection` class.

```
private static WsConnection s_WsCon;

public static void main(String[] args)
{
    try
    {
        // get property object
        Properties prop = WsUtility.getProperty(args[0]);

        // URL to content services web services servlet
        String serverUrl = "http://" + prop.getProperty("hostname") + ":"
            + prop.getProperty("port") + "/content/ws";

        // authenticate to content services
        s_WsCon = WsConnection.login(serverUrl, prop.getProperty("user"),
            prop.getProperty("password"));

        try
        {
            // log-in to user's personal workspace
            FileManager fm = s_WsCon.getFileManager();
            Item workspace = fm.resolvePath(prop.getProperty("userhome"), null);

            // create folders
            Item folder1 =
                createFolder("firstFolder", "This is the 1st folder", workspace);
            Item folder2 =
                createFolder("secondFolder", "This is the 2nd folder", workspace);

            // create document
            Item document =
                createDocument("myDoc", "This is an example doc", folder1);

            // move document to second folder
            moveDocument(document, folder2);

            // copy document to first folder
            copyDocument(document, folder1);
```

```
                // delete document
                deleteDocument(document);

                // delete objects
                cleanup(folder1, folder2);
            }
            finally
            {
                s_WsCon.logout();
            }
        }
        catch (Throwable t)
        {
            t.printStackTrace();
        }
    }


    /**
     * Delete created objects
     */
    public static void cleanup(Item folder, Item folder1)
        throws FdkException, RemoteException
    {
        // get the Manager instances
        CommonManager cm = s_WsCon.getCommonManager();
        FileManager fm = s_WsCon.getFileManager();
        TrashManager tm = s_WsCon.getTrashManager();

        // TRASH_FOLDER AttributeRequest array
        AttributeRequest[] trash_attr =
            WsUtility.newAttributeRequestArray(Attributes.TRASH_FOLDER);

        // get TRASH_FOLDER attribute for folder
        folder = cm.getItem(folder.getId(), trash_attr);
        NamedValue[] attrs = folder.getRequestedAttributes();
        Map attrMap = ClientUtils.namedValuesToMap(attrs);
        Item trashItem = (Item) attrMap.get(Attributes.TRASH_FOLDER);

        // delete the folder and empty the Trash
        fm.delete(new long[] { folder.getId() }, null, null);
        tm.emptyTrash(trashItem.getId());

        // get TRASH_FOLDER attribute for folder1
        folder1 = cm.getItem(folder1.getId(), trash_attr);
        attrs = folder.getRequestedAttributes();
        attrMap = ClientUtils.namedValuesToMap(attrs);
        trashItem = (Item) attrMap.get(Attributes.TRASH_FOLDER);

        // delete the folder and empty the Trash
        fm.delete(new long[] { folder1.getId() }, null, null);
        tm.emptyTrash(trashItem.getId());
    }
```

6 Attribute Requests

Attribute requests are a commonly used feature when developing for Oracle Content Services. Attribute requests tell Oracle Content Services to return a set of attributes about a specific `Item` in the repository. The request is given when performing an action on that `Item`, and is returned in an array of `AttributeRequest` objects. The examples in this book have all used basic attribute requests to retrieve simple piece of data. Attributes, however, can be nested, allowing the creation of more complex data structures for attribute requests. The following example shows how to request and retrieve a nested attribute structure.

6.1 Chaining Attribute Requests

In the case where attributes are composed of multiple values, attribute requests must be *nested* or *chained*. Chaining attribute requests allows retrieval of an entire tree of related objects in one call. Consider the following example, which sets version numbers on a folder:

```
/**
 * Set manual and auto version on folder
 */
public static void setVersion(Item folder)
    throws FdkException, RemoteException
{
    // get the Manager instances
    CommonManager cm = s_WsCon.getCommonManager();
    VersionManager vm = s_WsCon.getVersionManager();

    // create auto version configuration definition
    NamedValue[] autoVerDef = WsUtility.newNamedValueArray(new Object[][] {
        { Attributes.VERSIONING_CONFIGURATION_AUTO_LABEL, Boolean.TRUE },
        { Attributes.VERSIONING_CONFIGURATION_AUTO_VERSION, Boolean.TRUE },
        { Attributes.VERSIONING_CONFIGURATION_MAX_VERSIONS, new Integer(1000) },
        { Attributes.CONFIGURATION_FINAL, Boolean.FALSE },
        { Attributes.CONFIGURATION_ENABLED, Boolean.TRUE } });

    // set auto version configuration on the folder
    vm.setVersioningConfiguration(folder.getId(), autoVerDef);

    // The version attributes request
    // This requests the attributes of the folder, and nests
    // the attributes to be retrieved from the versioning configuration
    // of the folder
    AttributeRequest[] fdr_ver_attr = new AttributeRequest[] {
        WsUtility.newAttributeRequest(Attributes.PATH),
        WsUtility.newAttributeRequest(Attributes.DESCRIPTION),
    // nest the attributes of the versioning configuration
    WsUtility.newAttributeRequest(Attributes.VERSIONING_CONFIGURATION,
        WsUtility.newAttributeRequestArray(new String[] {
            Attributes.VERSIONING_CONFIGURATION_AUTO_LABEL,
            Attributes.VERSIONING_CONFIGURATION_AUTO_VERSION,
            Attributes.VERSIONING_CONFIGURATION_MAX_VERSIONS,
            Attributes.CONFIGURATION_ENABLED })) };

// requesting folder version attributes
// use the CommonManager to get the attributes of an Item
folder = cm.getItem(folder.getId(), fdr_ver_attr);

// returned item has the requested attributes in a NamedValue[]
// convert it to a map for easy lookup and access to attributes
NamedValue[] attrs = folder.getRequestedAttributes();
Map attrMap = ClientUtils.namedValuesToMap(attrs);
WsUtility.log("");

// attributes name and value
WsUtility.log(Attributes.PATH);
// PATH attribute
WsUtility.log(WsUtility.INDENT + (String) attrMap.get(Attributes.PATH));

WsUtility.log(Attributes.DESCRIPTION);
WsUtility.log(WsUtility.INDENT
    + (String) attrMap.get(Attributes.DESCRIPTION));

WsUtility.log(Attributes.VERSIONING_CONFIGURATION);
// VERSIONING_CONFIGURATION is again an Item which will have the nested
   attributes
// that we requested for
Item verItem = (Item) attrMap.get(Attributes.VERSIONING_CONFIGURATION);

// again we get these as NamedValue[]
// convert it to a map for easy lookup
attrs = verItem.getRequestedAttributes();
attrMap = ClientUtils.namedValuesToMap(attrs);

WsUtility.log(WsUtility.INDENT
    + Attributes.VERSIONING_CONFIGURATION_AUTO_LABEL);
WsUtility.log(WsUtility.INDENT + WsUtility.INDENT
    + (Boolean) attrMap.get(Attributes.VERSIONING_CONFIGURATION_AUTO_LABEL));

WsUtility.log(WsUtility.INDENT
    + Attributes.VERSIONING_CONFIGURATION_AUTO_VERSION);
WsUtility.log(WsUtility.INDENT
    + WsUtility.INDENT
    + (Boolean) attrMap.get(
        Attributes.VERSIONING_CONFIGURATION_AUTO_VERSION));

WsUtility.log(WsUtility.INDENT
    + Attributes.VERSIONING_CONFIGURATION_MAX_VERSIONS);
WsUtility.log(WsUtility.INDENT
    + WsUtility.INDENT
    + (Integer) attrMap.get(
        Attributes.VERSIONING_CONFIGURATION_MAX_VERSIONS));

WsUtility.log(WsUtility.INDENT + Attributes.CONFIGURATION_ENABLED);
WsUtility.log(WsUtility.INDENT + WsUtility.INDENT
    + (Boolean) attrMap.get(Attributes.CONFIGURATION_ENABLED));
```

```
}
```

The attribute request stored in the array `fdr_ver_attr` is an example of a nested attribute request. The attribute `VERSIONING_CONFIGURATION` contains multiple sub-attributes, all of which are stored in a subarray of the original request. Oracle Content Services can produce attributes for any level of nesting.

Reading nested attributes follows the same process as regular attributes. Since each nested attribute is itself a fully formed attribute request, you can use the `getRequestedAttributes` method on its `Item` object.

Note that the examples use the `WsUtility` class, which is included with the samples. This class contains many convenient wrappers for simple operations such as object instantiation and logging. These wrappers are used primarily for the sake of readability, and are not necessary.

Below is a second example, featuring two levels of nesting. The nested attribute request extracts attributes for a document, one of which is the nested `VERSIONS` attribute. For each version, the request fetches information about that versioned document, as well as the nested `VERSIONED_DOCUMENT` attribute. This attribute contains information about the versioned document itself. The entire structure is returned in the `docs` variable, converted to a Map, and extracted.

```
/**
 * Checks in a Document.
 */
public static void checkinDocument(Item workingCopy)
    throws FdkException, RemoteException
{
    // get the Manager instances
    CommonManager cm = s_WsCon.getCommonManager();
    VersionManager vm = s_WsCon.getVersionManager();

    // create checkin document definition
    // set a label and comment while checking in
    NamedValue[] checkInDef = WsUtility.newNamedValueArray(new Object[][] {
        { Attributes.VERSION_COMMENT, "a new version" },
        { Attributes.VERSION_LABEL, "2.0" } });

    NamedValueSet[] checkInDefs = new NamedValueSet[] {
        WsUtility.newNamedValueSet(checkInDef) };

    // check in the document
    vm.checkin(new long[] { workingCopy.getId() }, null, checkInDefs, null);

    // request for the attributes of the document
    // we look for the document's attributes,
    // attributes for different versions of this document
    // and for each version we get the details of the version
    AttributeRequest[] doc_ver_attr = new AttributeRequest[] {
        WsUtility.newAttributeRequest(Attributes.PATH),
        WsUtility.newAttributeRequest(Attributes.DESCRIPTION),
        // request for the version details
        WsUtility.newAttributeRequest(Attributes.VERSIONS,
            new AttributeRequest[] {
                WsUtility.newAttributeRequest(Attributes.VERSION_LABEL),
                WsUtility.newAttributeRequest(Attributes.VERSION_COMMENT),
                WsUtility.newAttributeRequest(Attributes.VERSIONED_DOCUMENT,
                    // request for attributes of each version
                    new AttributeRequest[] {
                    WsUtility.newAttributeRequest(Attributes.DESCRIPTION),
                WsUtility.newAttributeRequest(Attributes.DOCUMENT_LANGUAGE),
                WsUtility.newAttributeRequest(Attributes.CHARACTER_SET) })
}) };

// fetch the attributes via the CommonManager
Item docs = cm.getItem(workingCopy.getId(), doc_ver_attr);

// the returned item has the requested attributes in a NamedValue[]
// convert it to a map for easy lookup and access to attributes
NamedValue[] attrs = docs.getRequestedAttributes();
Map attrMap = ClientUtils.namedValuesToMap(attrs);
WsUtility.log("");

// the document's attributes
WsUtility.log(Attributes.PATH);
WsUtility.log(WsUtility.INDENT + (String) attrMap.get(Attributes.PATH));

WsUtility.log(Attributes.DESCRIPTION);
WsUtility.log(WsUtility.INDENT
    + (String) attrMap.get(Attributes.DESCRIPTION));

WsUtility.log(Attributes.VERSIONS);
// these are the different versions of the document
Item[] verItem = (Item[]) attrMap.get(Attributes.VERSIONS);
for (int i = 0; i < verItem.length; i++)
{
    // get second level of attributes
    attrs = verItem[i].getRequestedAttributes();
    attrMap = ClientUtils.namedValuesToMap(attrs);

    WsUtility.log(WsUtility.INDENT + Attributes.VERSION_LABEL);
    WsUtility.log(WsUtility.INDENT + WsUtility.INDENT
        + (String) attrMap.get(Attributes.VERSION_LABEL));

    WsUtility.log(WsUtility.INDENT + Attributes.VERSION_COMMENT);
    WsUtility.log(WsUtility.INDENT + WsUtility.INDENT
        + (String) attrMap.get(Attributes.VERSION_COMMENT));

    WsUtility.log(WsUtility.INDENT + Attributes.VERSIONED_DOCUMENT);
    // for each version the attributes of those documents that were versioned
    Item docItem = (Item) attrMap.get(Attributes.VERSIONED_DOCUMENT);

    // get third level of attributes
    attrs = docItem.getRequestedAttributes();
    attrMap = ClientUtils.namedValuesToMap(attrs);
    WsUtility.log(WsUtility.INDENT + WsUtility.INDENT
        + Attributes.DESCRIPTION);
    WsUtility.log(WsUtility.INDENT + WsUtility.INDENT + WsUtility.INDENT
        + (String) attrMap.get(Attributes.DESCRIPTION));

    WsUtility.log(WsUtility.INDENT + WsUtility.INDENT
        + Attributes.DOCUMENT_LANGUAGE);
    WsUtility.log(WsUtility.INDENT + WsUtility.INDENT + WsUtility.INDENT
        + (String) attrMap.get(Attributes.DOCUMENT_LANGUAGE));

    WsUtility.log(WsUtility.INDENT + WsUtility.INDENT
        + Attributes.CHARACTER_SET);
    WsUtility.log(WsUtility.INDENT + WsUtility.INDENT + WsUtility.INDENT
        + (String) attrMap.get(Attributes.CHARACTER_SET));
}
```

```
}
```

6.2 NamedValue and NamedValueSet

Every `Item` object has a `getRequested` Attributes method returns the attribute set in an array of `NamedValue` objects. The `NamedValue` class represents a name/value pair. In Oracle Content Services this class is used to represent attributes, options, and preferences. When working with a returned `NamedValue` array, it is often easier to convert it to a Map to allow lookups by attribute name. The preceding examples use the `namedValuesToMap` method (found in the included `ClientUtils` class), which returns a Map object comprising all the elements of the `NamedValue` array, with the name elements as key. Another option is to use the `NamedValueSet` class, which stores arrays of `NamedValue` objects. This class is used primarily to avoid the need to manage multi-dimensional arrays of attributes.

7

Uploading and Downloading Using Web Services

Oracle Content Services allows you to post and retrieve documents between a client and a Web Services instance using the DAV (Distributed Authoring and Versioning) protocol.

For this example, you will need the following managers:

- File Manager - For creating a file on the Web Services instance when uploading, and resolving folder names
- Session Manager - for maintaining the Manager sessions

You will also need to use the `HTTPConnection` class to connect to the DAV server when uploading and downloading. This class is included as part of the iAS `HTTPClient` library.

Uploading

The first step in uploading a document to the server is to create a new DocumentDefinition `Item` on the server. To do this, use the `FileManager.createDocumentDefinition()` method, which creates a `DOCUMENT_DEFINITION Item` on the server. This `Item` acts as a placeholder for a Document prior to uploading its contents. Pass an `AttributeRequest` to the FileManager method call to retrieve the fully resolved URL (`Attributes.URL`) of this Document. This URL is later used to upload the content.

```
/**
 * Creates a new Document with the specified content.
 */
public static Item uploadDocument(String folderPath, String docName,
    byte[] content) throws FdkException, IOException, ModuleException
{
    // get the Manager instance
    FileManager fm = s_WsCon.getFileManager();

    // create an AttributeRequest[] for the URL attribute
    AttributeRequest[] urlAR =
        WsUtility.newAttributeRequestArray(Attributes.URL);

    // create a DOCUMENT_DEFINITION Item
    // -> a temporary, persistent document definition into which content
    //    can be uploaded using HTTP
    // -> notice that we request the URL attribute for the Item that is
    //    returned; we will use the URL to upload contents to the docDef
    Item docDef = fm.createDocumentDefinition(
    WsUtility.newNamedValueArray(new Object[][] {
        { Attributes.NAME, docName } }), urlAR);
```

Now that the placeholder document exists, open an HTTP connection to it using the URL returned by the `AttributeRequest`, and transfer the data using the `HTTPConnection.put()` method.

For convenience, this section uses the `ClientUtils` class, which is a set of data type operators and utility methods for Web Services clients. To extract the URL attribute, a `Map` is created from the requested attribute set using the `ClientUtils.namedValuesToMap` method, and the URL retrieved using the `Map.get` method. This technique offers the advantage of being able to retrieve the attributes by name, instead of having to iterate over the array.

This example also uses a helper method called `createHttpConnection()`, which is explained at the end of this section.

```
    // get the URL attribute from the docDef Item
    NamedValue[] attrs = docDef.getRequestedAttributes();
    Map attrMap = ClientUtils.namedValuesToMap(attrs);
    String docUrl = (String) attrMap.get(Attributes.URL);
    URL url = new URL(docUrl);

    // upload content into docDef using HTTP
    HTTPConnection httpCon = createHttpConnection(url);
    HTTPResponse rsp = httpCon.Put(url.getFile(), content);

    // check the response
    if (rsp.getStatusCode() >= 300)
    {
        System.err.println("Error: " + rsp.getReasonLine());
    }
```

Once the content is uploaded, create the final document using the `FileManager` by passing in the destination folder and the document definition. The `USE_SAVED_DEFINITION` option instructs the `FileManager` to use the newly created document definition (`docDef`) as the basis for this new document.

```
    // create the Document using the docDef
    Item folder = fm.resolvePath(folderPath, null);
    Item newDoc = fm.createDocument(
    WsUtility.newNamedValueArray(new Object[][] {
        { Options.DESTFOLDER, new Long(folder.getId()) },
        { Options.USE_SAVED_DEFINITION, new Long(docDef.getId()) } }), null,
            urlAR);

    return newDoc;
}
```

Now, the helper method:

```
/**
 * Creates an HttpConnection ready to be used for uploading and downloading.
 */
private static HTTPConnection createHttpConnection(URL url)
    throws ProtocolNotSuppException
{
    // create the HTTPClient cookie, using the session cookie
    //    -> we specify "/content" as the cookie path because
    //          it is always first in Oracle Content Services URLs
    String[] sessionCookies = s_WsCon.getSessionCookie();
    String sessionCookie = sessionCookies[0];
    Cookie cookie = new Cookie("c1", sessionCookie, url.getHost(), "/content",
                        null, false);

    // create a context object and add the cookie to it
    Object ctx = new Object();
    CookieModule.addCookie(cookie, ctx);

    // create an HttpConnection to the Document and set its context
    HTTPConnection httpCon = new HTTPConnection(url);
    httpCon.setContext(ctx);

    // turn off interactive mode
    httpCon.setAllowUserInteraction(false);

    return httpCon;
}
```

To upload a file, you must create a cookie for the given domain and path in order to keep the session alive while transferring data. Since you will be using the Java `HTTPConnection` library to upload the file, pass the cookie and a new context object to the `CookieModule`. This context will be associated with the session, and can then be passed to the `HTTPConnection` to maintain session information.

Finally, open a connection to the URL, pass the context to the connection, and turn off user interaction to prevent any HTTP user prompts. The returned `HTTPConnection` is then used to make a PUT call to pass the contents into the remote file.

Downloading

The process for downloading a file is similar to uploading, except that you must use the `HTTPConnection.get()` method. Also, there is no need to create a document; simply download the content into an `HTTPResponse` object, and pass the downloaded bytes into a stream.

```
/**
 * Downloads and prints the document content.
 */
public static void downloadDocument(Item doc)
    throws FdkException, IOException, ModuleException
{
    // get the URL attribute from the docDef Item
    NamedValue[] attrs = doc.getRequestedAttributes();
    Map attrMap = ClientUtils.namedValuesToMap(attrs);
    String docUrl = (String) attrMap.get(Attributes.URL);
    URL url = new URL(docUrl);

    // download the document content using HTTP
    HTTPConnection httpCon = createHttpConnection(url);
    HTTPResponse rsp = httpCon.Get(url.getFile());

    // check the response
    if (rsp.getStatusCode() >= 300)
    {
        WsUtility.log("Error: " + rsp.getReasonLine());
    }
```

```
        // write downloaded content in the file
        WsUtility.log("write downloaded content in the file:");
        InputStream content = rsp.getInputStream();
        FileOutputStream fos = new FileOutputStream(s_Prop.getProperty("outputfile"));
        int c = -1;
        while ((c = content.read()) != -1)
        {
            fos.write(c);
        }
    }
```

Running the Code

To run the code, connect to an Oracle Content Services instance and call the upload and download methods. Remember to log out of the Web Services instance using the `logout()` method of the `WsConnection` class.

```
private static WsConnection s_WsCon;

public static void main(String[] args)
{
    try
    {
        // get property object
        s_Prop = WsUtility.getProperty(args[0]);

        // URL to content services web services servlet
        String serverUrl = "http://" + s_Prop.getProperty("hostname") + ":"
            + s_Prop.getProperty("port") + "/content/ws";

        // authenticate to content services
        s_WsCon = WsConnection.login(serverUrl, s_Prop.getProperty("user"),
            s_Prop.getProperty("password"));

    try
        {
            // read the content from file to upload
            FileInputStream fis = new FileInputStream(
                                        s_Prop.getProperty("inputfile"));
            int len = fis.available();
            byte[] content = new byte[len];
            fis.read(content);

            // create a document using the content
            Item doc = uploadDocument(s_Prop.getProperty("userhome"),
                                         "testDoc.txt",
                                          content);

            // download and show the document's contents
            downloadDocument(doc);

            // delete objects
            cleanup(doc);
        }
        finally
        {
            s_WsCon.logout();
        }
    }
    catch (Throwable t)
    {
        t.printStackTrace();
    }
}
```

The helper method for cleaning up method is listed below.

```
/**
 * Deletes created objects.
 */
public static void cleanup(Item document)
    throws FdkException, RemoteException
{
    // get the Manager instances
    CommonManager cm = s_WsCon.getCommonManager();
    FileManager fm = s_WsCon.getFileManager();
    TrashManager tm = s_WsCon.getTrashManager();

    // TRASH_FOLDER AttributeRequest array
    AttributeRequest[] trash_attr =
        WsUtility.newAttributeRequestArray(Attributes.TRASH_FOLDER);

    // get TRASH_FOLDER attribute
    document = cm.getItem(document.getId(), trash_attr);
    NamedValue[] attrs = document.getRequestedAttributes();
    Map attrMap = ClientUtils.namedValuesToMap(attrs);
    Item trashItem = (Item) attrMap.get(Attributes.TRASH_FOLDER);

    // delete the document and empty the Trash
    fm.delete(new long[] { document.getId() }, null, null);
    tm.emptyTrash(trashItem.getId());
}
```

8 Oracle Content Services Versioning

Oracle Content Services provides a way to *version* items. Versioning tracks changes made to the content and metadata of an item throughout its life cycle, and allows users to see and revert to older versions of the item.

Versioning in Oracle Content Services

Oracle Content Services allows an administrator to choose the right balance between user requirements, system resources, and performance. Tracking every version of every document in the system can stress the system's resources, so it is important for administrators to be able to control the specifics of the versioning configuration. To allow better control, Oracle Content Services allows customizing of the versioning policy at the site, Library, Container, or Folder level.

Versioning Policies

Oracle Content Services can create a new version of a document whenever a modifying operation is performed on an existing document such as an `FileManager.updateDocument` call, upon creation of a new document (with a `FileManager.createDocument` call), or as part of a copy or move operation (with a `FileManager.copy` or `move` call).

Versioning policy can be set for the entire site, or it can be set at a lower level, such as on a Library, Container, or folder. There are three versioning strategies: *automatic*, *manual*, and *none*.

With automatic versioning enabled, performing an operation that would cause an existing document to be modified (such as a copy, move, update, or create operation) will create a new version of that document. If the existing document was not versioned, two versions of the document will exist. Otherwise, the version count on the document will increment by 1. Any `createDocument` call requires write access on the destination, and the destination cannot be locked (for example, the document could be locked if explicitly checked out by another user, or it could be locked as part of a workflow process). With automatic versioning enabled, explicit checking-out of a document is optional.

With manual versioning enabled, changing a document in the system will fail unless the document has been explicitly checked out by the calling user. If a different user has checked out the document, you will have to wait until the document becomes available. The `VersionManager.checkoutDocuments` method allows you to check out one or more documents, providing you have permissions on the folder to do so.

Versioning can also be disabled entirely. In this case, any operation that would cause an existing document to be modified (such as a copy, move, update, or create operation) will overwrite the existing document's content and metadata.

Versioning Configuration

The default versioning policy for an Oracle Content Services instance is "no versioning". The versioning policy can be set at the site, Library, Container, or folder level, and all its descendants will inherit the policy by default. A descendant can override the inherited versioning policy only if the `CONFIGURATION_FINAL` attribute was set to false on the original policy. To prevent descendants from overriding the versioning policy, set that policy's `CONFIGURATION_FINAL` attribute to true.

Setting the versioning policy at the site level must be done in administrator mode, whereas setting the policy on a Library, Container, or folder only requires the appropriate permissions on the object.

The versioning policy also dictates the maximum number of versions to store, by setting the `VERSIONING_CONFIGURATION_MAX_VERSIONS` attribute. Any updates to the version count will be checked against this number, and when exceeded, the oldest stored version will be deleted. However, if a version is set with the `DO_NOT_PURGE` flag set to true, it will not count against the maximum version count. In other words, `VERSIONING_CONFIGURATION_MAX_VERSIONS` refers to the number of versions to keep which have `DO_NOT_PURGE` set to false. These criteria typically refer to versions created by an automatic versioning policy. The `DO_NOT_PURGE` flag of a version can be changed at any time with the `updateVersion` method of the VersionManager.

Versions are assigned *comments* and *labels.* Comments describe the version, and the label indicates the version number. The `VERSIONING_CONFIGURATION_LABEL_TYPE` attribute determines what kind of label a version gets (numbers, letters, or Roman numerals). By default, Oracle Content Services will assign the labels automatically according to the label type. To change how labels are assigned, set the `VERSIONING_CONFIGURATION_AUTO_LABEL` attribute to false.

Rolling back to an older version of a document involves copying the older version onto the latest version using the `VersionManager.copyToLatestVersion` method. Similarly, to download a working copy of any version in the item's version history, use the `copyToWorkingCopy` method, and supply the ID of the version.

Conflict Resolution

When uploading new of versions files which have previously been versioned, the possibility for a naming or versioning conflict arises. Oracle Content Services offers two ways of handling a conflict.

The first way is to set options on the file describing the resolution policy at the time of its creation. These options are passed inside the `createDocument` method of the FileManager. The options are:

- `Options.NEWVERSION` - this option ensures that when uploading to a folder with manual or automatic versioning configuration, a new version will always be created on upload. There is no need to explicitly check out the document before uploading.
- `Options.UNIQUENAME` - this option ensures that each upload will produce a unique name, so that no file will ever be overwritten. Thus, attempting to upload an existing filename will result in a slightly modified new filename.

- `Options.OVERWRITE` - this option ensures that when uploading to a folder with no versioning policy, any existing item with the same name will automatically be overwritten.

The other way of handling a conflict is by allowing the `createDocument` call to throw an exception, and processing the conflict based on the returned error. This strategy allows more user interaction as the application can then defer to the user to provide the correct action.

A sample exception is listed below, with the relevant error messages highlighted:

```
FdkException:
    Error Code: ORACLE.FDK.AggregateError
    Detailed Error Code: ORACLE.FDK.AggregateError
    Trace Id: 33-1138800897129
    info (NamedValue[]):
    FdkExceptionEntry:
    Id: 86842
    Error Code: ORACLE.FDK.OperationError
    Detailed Error Code: ORACLE.FDK.ItemAlreadyExists
    Trace Id: 32-1138800897126
    info (NamedValue[]):
    ECM.EXCEPTIONINFO.ConflictResolutionOptions (String[])=
        OPT.NEWVERSION
        OPT.UNIQUENAME
    ECM.EXCEPTIONINFO.ConflictingObjectId=86836 (Long)
```

The `ConflictResolutionOptions` array of the exception returns the names of the possible conflict resolution options. Retrying the operation with one of these options set on the transaction will increase the chance of (but not necessarily guarantee) the success of the transaction.

> **Tip:** Since the conflict resolution takes place at the end of the file transaction, a user uploading a large file to the server may be forced to upload the file again if there is a versioning conflict. To avoid this, use a DocumentDefinition item (created by the `FileManager.createDocumentDefinition` method) when uploading so that the content of the file remains on the server even if there is a conflict. After the content is uploaded, call `createDocument` to create the file object and resolve the conflict.

9

Oracle Content Services Web Services Managers

Oracle Content Services Web Services consists of about two hundred operations that can be remotely invoked. Related operations are organized into categories called managers. A manager has its own WSDL file and Java interface in the Web Service Java stubs library.

This chapter describes the following Oracle Content Services Web service managers:

- ArchiveManager
- CategoryManager
- ContainerManager
- DomainManager
- FileManager
- GroupManager
- LockManager
- PagingManager
- QuotaManager
- RecordsManager
- RemoteLoginManager
- RequestManager
- SearchManager
- SecurityManager
- ServiceToServiceManager
- SessionManager
- SortManager
- TrashManager
- UserManager
- VersionManager
- VirusManager
- WorkflowManager
- WorkspaceManager

Reference Material

The complete listing of service managers along with their supported operations can be found at the URL http://<*host*>:<*port*>/content/ws with <host> and <port> values replaced to reflect that of your own Oracle Content Services 10g instance.

Descriptions of individual operations and the parameters required for each operation are located in the Oracle Content Services Web Services Java API Reference (Javadoc) of the interface of the same name or the class that has implemented that interface.

Descriptions of the XML structure of each of a manager's operations are located in the WSDL file named after the manager at the URL http://<*host*>:<*port*>/content/wsdl/<*name of manager*>.wsdl.

Document Managers

These managers provide operations for manipulating files and folders.

FileManager

The FileManager provides core document, folder, and link management capabilities.

The FileManager supports the following operations:

- Resolve an item based on a supplied path string (the returned Item will be a Document, Link, or Folder)
- Resolve an item based on a supplied relative path string and source folder
- Check to see whether an item exists based on a supplied absolute path
- Check to see whether an item exists based on a supplied relative path string and source folder
- Create and return folders in the supplied destination folders
- Update a folder
- Return (list) items in a given folder, with optional sort criteria
- Check to see whether an object with the given name can be created in the given folder
- Copy one or more specified items to a given destination folder
- Move one or more specified items to a given destination folder
- Delete one or more specified items (deleting a folder will delete of all its items)
- Create one or more document definitions. (Note: These definitions are created with empty content; to load content, perform an HTTP PUT to the WebDAV Server.)
- Create one or more documents, potentially using saved document definitions as source. (Note: These documents are created with empty content; to load content, perform an HTTP PUT to the WebDAV Server.)
- Update a document
- Create a link to a given item in specified destination folder (links can be made to items of type: Domain, Container, Workspace, Folder, Document, and Family)
- Update a link
- Check if a given path string contains any link items

- Get supported languages
- Get supported character sets
- Return most recent documents for connected user, with optional sort criteria (the MostRecentDocAgent is responsible for maintaining users' most recent document statistics)
- Uncompress supplied items
- Get name conflict resolution options
- Create documents, document definitions, folders and links
- Get a list of supported languages or character sets

Note: Content upload/download is handled by standard HTTP PUT/GET, not Web service attachments.

TrashManager

The TrashManager provides trash folder operations. All workspaces (including personal workspaces) come with a trash folder that, if enabled, stores deleted items.

The TrashManager supports the following operations:

- Empty the specified trash folder
- Configure the specified trash folder. This includes enabling or disabling it, enabling or disabling the auto-empty feature, and setting the minimum holding period before the trash folder purges itself (when the auto-empty feature is enabled).

ArchiveManager

The ArchiveManager provides archive operations. When a trash folder is emptied, its contents can be moved to a special area in the repository known as an archive. Each domain has its own archive that is accessible by content administrators.

The ArchiveManager supports the following operations:

- Empty an archive
- Restore an item from the archive, or make a request to a content admininstrator to restore an item
- Configure an archive. An archive can be enabled or disabled, set to automatically empty itself, and have a minimum holding period for its contents.

Document Processing Managers

These managers handle how documents are listed, accessed, and categorized.

SortManager

The SortManager sets user's default sort preferences for the following tables used by the Oracle Content Services application. These tables are defined in the `FdkConstants` class in the Web services Java API. These tables include:

- fileListingTable
- versionHistoryTable
- joinableWorkspacesTable

- lockedFilesTable
- userSearchResultsTable
- checkedOutFilesTable
- workflowReportTable
- virusNamesTable
- recentFilesTable
- groupMemberList
- groupMembers
- memberListingTable
- rolesTable
- securityConfigTable
- selectedUsersOrGroupsTable
- usersOrGroupsTable

In addition, many Web services operations that return item arrays support setting the sort sequence as part of their operation parameters.

The SortManager supports the following operations:

- Set the user's sort preference for the specified table
- Obtain the user's primary or secondary sort attribute for the specified table
- Obtain the user's primary or secondary sort direction (ascending or decending) for the specified table
- Sort an array or a list of items (where the array of items are an attribute of the specified item)

SearchManager

The SearchManager provides the ability to search for documents, with the use of a SearchExpression tree. This is a complex type object that consists of two operands, left operand and right operand, which are associated by a specified operator. Depending on the type of operator, the left and/or right operands may themselves be SearchExpression nodes. This enables one to build a complex SearchExpression tree. The following table describes the operators and supported by SearchExpression:

Table 9–1 Operators Supported by SearchExpression

Operator	Syntax	Notes and Examples

EQUAL
GREATER_ THAN
GREATER_ THAN_EQUAL
LESS_THAN
LESS_THAN_ EQUAL
NOT_EQUAL
IN
	Left operand is the string name of the attribute being compared. Right operand is the value being compared, represented as a String, Integer, Long, or Date.	Wildcard characters "*" and "?" are supported only for the EQUAL comparison operator so long as comparing an attribute with datatype String.
		Example:
		`SIZE GREATER_THAN 1048576`
		`NAME EQUAL *.doc`

Table 9–1 (Cont.) Operators Supported by SearchExpression

Operator	Syntax	Notes and Examples
CONTAINS	Left operand must be null, Right operand specifies words or phrases to be found in content of a document.	Words are specified by spaces, and phrases are enclosed in double quotes ("). Example: `<NULL> CONTAINS "Content Services"`
AND	Left and Right operands must themselves be SearchExpressions	`(SIZE GREATER_THAN 1048576) AND (NAME EQUAL *.doc)`
OR	Left and Right operands must themselves be SearchExpressions	`(SIZE GREATER_THAN 1048576) AND (NAME EQUAL *.doc)`
NOT	Left operand must be null, Right operand must be a SearchExpression	`<NULL> NOT (((SIZE GREATER_ THAN 1048576) AND (NAME EQUAL *.doc))`

A Search is invoked by supplying SearchManager's search operation with a SearchExpression tree along with zero or more optional search options.

The SearchManager supports the following operations:

- Restrict search to one or more specified folders (including optionally sub-folders)
- Include or exclude non-current versions of a versioned document from being searched
- Set the start index for the first item in the server's search result array that should be returned. (Default is 1, which means to return all items from the search result array starting from the very first search result)
- Limit the number of items returned back to the client from the server's search result array

LockManager

The LockManager provides operations for working with Locks, which prevent outside changes from occurring on an item that is currently being worked on. The repository will automatically deploy certain types of locks transparently when a caller requests specific types of operations (for example, a checkout call). Certain client applications such as Microsoft Office that have native WebDAV support often will also request a DAV lock during the course of editing a supported document type.

The LockManager supports the following operations:

- Acquire a manual lock on one or more items
- Release a manual lock on one or more items
- Return a list of items locked by the current user that match the specified lock type(s), for example, manual lock or WebDAV lock.

Lock Type Constants from oracle.ifs.fdk.FdkConstants	Value
LOCKTYPE_MANUAL	1
LOCKTYPE_CHECKOUT	2
LOCKTYPE_FINALIZED	3
LOCKTYPE_WORKFLOW	4

Lock Type Constants from oracle.ifs.fdk.FdkConstants	Value
`LOCKTYPE_RECORD`	5
`LOCKTYPE_FAMILYHASRECORD`	6
`LOCKTYPE_DAV`	7

VersionManager

The VersionManager tracks changes made to an item's content and metadata throughout its lifecycle. Tracking every revision of every document in the system is expensive from a system resources perspective, though potentially a requirement for certain businesses. However, Oracle Content Services allows an administrator to choose the right balance between user requirements, system resources, and performance.

A folder can have the following versioning configuration settings:

- Whether manual or automatic version should be applied to items in the folder
- If automatic versioning is being used, whether to enforce a limit on the number of revisions maintained
- Whether the versioning configuration is final: subfolders cannot override the versioning configuration

Oracle Content Services uses a serial versioning model. The server maintains a single version series for each versioned document. To manually create a new version of a document, the following steps occur:

1. The author checks out the document.
2. The server makes a working copy of the latest version (including both content and metadata). This server-resident working copy is accessible only to the user who checked out the document.
3. A lock is also issued to prevent other authors from checking out the versioned document.
4. When the author is finished making changes to the working copy, he or she checks in the document.
5. A new version of the document is created. The new version becomes the latest version of the document, and like any document version, is immutable and cannot be further updated.
6. The lock acquired at check-out is then released, allowing other users to check-out the document and the working copy object is destroyed.

The VersionManager supports the following operations:

- Check out a set of items
- Cancel check out for a set of items
- Copy a specified version to another folder, to the working copy, or as the latest version
- Move a specified version to another directory
- Delete a specified version so that it is no longer part of the version histroy
- Retrieve an item’s versions

- Check in a set of items
- Update a version-controlled item's attributes (such as version comments, the version label, and the `do_not_purge` flag)
- Make a non version-controlled document versioned
- Replace or remove the versioning configuration of a folder

VirusManager

The VirusManager scans and potentially repairs documents for viruses. Oracle Content Services ships a default virus scanning adapter that uses Symantec AntiVirus Scan Engine (SAVSE) using ICAP 1.0.

An asynchronous background agent, VirusScanAgent, regularly polls the virus scanning adapter to determine if a new virus definition build is available. When a new virus definition is detected, the system domain property `IFS.DOMAIN.VIRUSSCANNER.LastVirusDefinitionUpdate` is modified accordingly.

The scanning process is on-demand. The following events occur when a document is requested:

1. The system looks at the metadata associated with that document to determine whether a virus scan needs to be performed. Associated with each document is a `LAST_SCANNED_DEFINITION_DATE` attribute, which tracks the virus definition timestamp that was last used to scan the document. If this attribute is null or older than the LastVirusDefinitionUpdate domain property, then the document's contents are scanned. (Otherwise, the document contents are immediately returned.)
2. Depending on the result of the scan, the document's contents are either returned to the user if a virus have been detected (or a repair was successful), or else an error is returned and the document is quarantined. The `LAST_SCANNED_DEFINITION_DATE` is also updated to reflect the new virus definitions that were utilized.

The following events occur when a document is quarantined:

1. The document's `IS_QUARANTINED` and `QUARANTINED_DATE` attributes are updated.
2. VirusScanAgent receives an event alerting it of the infected document.
3. The agent attempts to repair quarantined documents that have a `REPAIR_ATTEMPTS` value less than the domain property `IFS.DOMAIN.VIRUSSCANNER.MaxRepairAttempts` and `LAST_SCAN_DEFINITION_DATE` older than the LastVirusDefinitionUpdate domain property.
4. The agent creates a VirusReport (Category Instance) that is associated with the quarantined document after the repair attempt.

Documents under quarantine have the following properties and behaviors:

- Document objects themselves will be unaffected by quarantine status (in particular, metadata can still be viewed or modified)
- Contents cannot be opened for read access under any circumstances. Attempts will result in an exception. Contents will remain unreadable even if the anti-virus option is disabled.
- Contents may be overwritten.
- Documents and their contents may be deleted.

- Documents will not have specific infection information available until the VirusScanAgent attempts to repair it.

The VirusManager supports the following operations:

- Scan specified items
- Attempt to repair specified items
- Retrieve the virus report for a specified item that was sent for repair
- Retrieve the currently known timestamp of the last virus definition update

RecordsManager

The RecordsManager provides a way to define the lengh of time certain documents should be stored, and how to destroy these documents after this length of time, or retention period, has elasped.

This manager provides the following operations:

- Create a file plan
- Create a record series under the specified file plan
- Create a record category under the specified record series or file plan
- Create a record folder under a record category
- Delete a file plan (the delete will fail if there are record series or record categories under it)
- Delete a record series (the delete will fail if there are record categories under it)
- Delete a record category (the delete will fail if there are any record folders or records under it)
- Delete a record folder (only empty record folders can be deleted)
- Update a file plan
- Update a record series
- Update a record category
- Update a record folder
- List file plans
- List record series under a specified file plan
- List record categories under a specified record series or file plan
- List record folders under a record category
- Get a file plan that matches a specified name
- Get a record series that matches a specified name under a specified file plan
- Get a record category that matches a specified name under a specified file plan or record series
- Get a record folder that matches a specified name under a specified record category
- Add a record category attribute
- Modify a record category attribute
- Return direct children of a record management object

- Make a record
- Update a record
- Get a recordized object given a record id
- Unrecordize a record from a given record category or record folder
- Set record configuration for a folder
- Delete record configuration for a folder
- Get a required record category for a specified folder

The RecordsManager uses the following items:

- **File Plan**: Document containing the disposition authority of a set of Records. A File Plan may contain Record Series and Record Categories.
- **Record Series**: Named container for a set of Record Categories.
- **Record Category**: Description of a set of Records within a File Plan. Each Record Category has retention and disposition data associated with it that is applied to all Record Folders and Records within it.
- **Record Folder**: Extension of a Record Category used to aggregate Records by cut-off date.
- **Record**: Information, regardless of medium, controlled by a particular Record Category.

PagingManager

The PagingManager allows control over pagination of items. When dealing with large item arrays (such as the result of a search), certain clients may find it is more convenient to deal with just a single "page" of items at a time. This approach is often favored by users who have low network bandwidth.

The PagingManager supports the following operations:

- Store a list of items in a paging list
- Retrieve a page of items of a specified size from the paging list at a specified start index

Note that there is only one paging list item array per session.

CategoryManager

The CategoryManager provides functions for configuring categories. Publicly accessible repository objects such as Document and Folder can store traditional file system metadata such as `description`, `create_date`, `created_by`, `last_modified_date`, and `last_modified_by`. Custom file system metadata is represented in Oracle Content Services by category objects.

Attributes hold the data that describes items in Oracle Content Services. Categories contain and organize attributes, as well as contain other categories, or category subclasses. When a specified category is applied to a document, for example, that document will contain the attributes of that category, as well as the attributes of the parent, or superclass, category.

A category configuration may be applied to a folder. This specifies which categories are required and allowed for items in that folder. It also specifies whether or not subfolders may override the configuration, and whether or not versioning is enabled.

For example, consider a root category called **Project Category**. This category contains an attribute called **Project Number** and another category called **Project Document Category**. This category subclass contains three attributes called **Billable Material**, **Document Type**, and **Document Reviewer**. For each attribute, you can define its type, whether or not it is required, a default value, and whether or not its default value can be overridden. For example, Billable Material can be a boolean value whose value is required, its default value "true", and whose default value can be overridden.

If an instance of Project Document Category is applied to an item, that item will have the attributes of Project Document Category as well as the attributes from Project Category.

The CategoryManager supports the following operations:

- Create, update, and delete categories and instances of categories
- Retrieve a list of all categories, or category subclasses of a specified category
- Retrieve, add, modify, and remove attributes from a category.
- Retrieve, apply, modify, and remove a category configuration for a specified folder

User and Group Managers

These managers handle users and groups.

UserManager

The UserManager supports the following operations:

- Get a user with a specified name
- Get and set user preferences for the current user with the specified preference keys
- Get and set domain-level (site-level) default user preference values
- Find provisioned users in the domain matching the specified search criteria
- Find users in the current user's Oracle Internet Directory (OID) that match the specified search criteria
- Synchronize the preferences of the current or specified user with those found in Oracle Internet Directory
- Search for a list of provisioned users in the current domain
- Search for a specific user by name

GroupManager

The GroupManager allows configuration of groups. A group is a collection of users and groups that can be managed as a single unit. Groups allow you to conveniently and quickly assign privileges to a collection of users without having to assign them to each user individually. Oracle Content Services groups are locally managed application objects and distinct to OID groups, which are not directly supported in this release. You can, however, manually provision OID groups as Oracle Content Services groups by using the GroupManager Web service.

Two distinct lists are maintained within a Oracle Content Services group: the member list, and the manager list. Managers may add and remove members and other managers from the group, and rename or delete the group.

The GroupManager supports the following operations:

- Search for, create, update, and delete groups

- Add and remove members and managers from a group

SecurityManager

The SecurityManager allows granting and revoking of roles to users and groups. A role is a set of access permissions that provide a user or a group privileges to perform certain operations. Chapter A, "Oracle Content Services Roles" provides the list of roles and their explicit permissions that ship out-of-the-box (OOTB) with Oracle Content Services. Two types of roles exist: core administration roles and standard (non-administration) roles. Some roles can be granted at a domain level, while other roles can be granted at the container or workspace level.

The SecurityManager supports the following operations:

- Retrieve, delete, create, and update roles
- Retrieve the available roles in the domain that apply to a specified item or item type
- Update, add grants to, or remove an item's security configuration. A security configuration is the set of all roles granted on a specific item (such as a workspace).
- Retrieve the set of users that are granted a specified role on a specified security configuration.
- Check whether or not the specified user or group has a specified permission *or* a specified role on a specified target item

Collaborative Managers

These managers help organize items in Oracle Content Services.

WorkspaceManager

The WorkspaceManager allows creation and configuration of workspaces, which are special folders created in a Container folder to store content such as documents and regular folders. Workspaces differ from containers and regular folders; they have a trash folder and can have an associated quota. Two types of workspaces are available: personal workspaces and shared workspaces. Personal workspaces are a special type of workspace for exclusive use by a user. Shared workspaces can be used by any user who is granted the appropriate privileges.

Workspaces must be uniquely named only within their parent container, while workspaces in different containers may have the same name. Therefore, the correct way to look up a workspace is by its path or unique ID.

The WorkspaceManager supports the following operations:

- Create or delete a workspace
- Update a workspace's name, description, or joinable flag
- List joinable workspaces
- Request to join a specified workspace

ContainerManager

The ContainerManager allows creation and configuration of containers, which are special folders in the system from which workspaces (and sub-containers) can be created. Containers cannot contain documents, only other containers and workspaces.

They have no quota and no trash. Containers can be created either directly under the domain, or under a parent container object.

The ContainerManager supports creating, deleting, and updating containers.

WorkflowManager

The WorkflowManager allows configuration of workflows. For a given item (such as an instance of a document or folder), a number of operations appropriate for the item's type can be configured to be controlled by the workflow. The list of available operations that can potentially be workflow controlled includes: createDocument, checkin, copy, delete, move, createWorkspace, joinWorkspace, and increaseQuota.

A workflow configuration is used to associate a particular item instance and operation type with a workflow item. A workflow configuration can be set to triggered (non-blocking) or approval-based (blocking). A triggered configuration means that the registered operation will occur right away (as if it were not workflow-driven), but a workflow process will still be started. An approval-based configuration means that the registered operation will not occur until the workflow process is approved.

An example of using a triggered workflow could be to capture all createDocument (upload) requests on a folder item and use logic in the workflow to respond to this event, such as calling a Web service, or logging the createDocument request to a file. An example of using an approval-based workflow could be to capture all delete requests and require that a particular responder approve the deletion of the particular item(s).

The WorkflowManager supports the following operations:

- return all registered workflow instances
- set a workflow configuration on a given item for the given operation type
- remove a workflow configuration from a given item for the given operation type
- retrieve all workflow configurations that exist for the given item
- retrieve workflow configuration for the given item that matches the given operation type
- validate that value of given workflow parameter is valid for the given workflow

DomainManager

The DomainManager allows configuration of Oracle Content Services domains. Every instance may have one or more domains (or sites). Each domain is an organizational entity that contains users and their content, metadata, and business rules.

The DomainManager supports obtaining the default domain and updating settings for a specified domain.

QuotaManager

The QuotaManager limits the amount of content that can reside in a Workspace (including users' personal workspaces).

The QuotaManager supports the following operations:

- Update the amount of allocated quota for a specified item
- Request an update to allocated quota for a specified item
- Calculate consumed quota for a specified item

Administrative Managers

These managers handle connections between clients and Oracle Content Services.

RemoteLoginManager

The RemoteLoginManager provides session creation and logout capabilities.

The RemoteLoginManager supports authenticating a user based on username and password, and disconnecting the current user session.

SessionManager

The SessionManager manages session information when performing transactions. When a client performs an operation in Oracle Content Services using Web services, such as creating, modifying, or deleting an object, the changes are immediately committed to the repository. However, in some cases the client may need to commit a set of dependent operations together; the client may need to ensure that if all these dependent operations cannot be successfully performed, none of these operations will not be performed. In other words, these operations must be atomically committed or rolled back. SessionManager provides methods that enable clients to wrap operations in a transaction block that can be used to control when the changes are committed to the repository.

The SessionManager supports the following operations:

- Return the current user connected as an item
- Switch the current session in and out of administration mode
- Begin, commit, or rollback a specified transaction during this session
- Keep alive a session to prevent it from timing out
- Get supplied session properties (such as `LOGIN_USER`, `SESSION_TIMEOUT`, `TRANSACTION_TIMEOUT`)

Oracle Oracle Content Services supports nested transactions. Consider the following example:

```
1 Begin Outer-Transaction
2    Begin Inner-Transaction
3        perform some operation
4        Begin Inner-Transaction
5            perform some operation
6        Abort (Rollback) Inner-Transaction
7    Commit Inner-Transaction
8 Abort (Rollback) Outer-Transaction
```

The innermost transaction (lines 4-6) is rolled back. Hence, any changes made in that particular transaction will never be applied. The encapsulating inner transaction (lines 2-7) is set to be committed, though its changes may be rolled back by the outer transaction. Because the outer transaction (line 8) is rolled back, any changes set to be committed in the inner transactions will not be applied. Thus, inner transactions are essentially save-points. They can be rolled back individually, but in order for them to be committed, the outer transaction must be committed.

RequestManager

The RequestManager allows operations on an item (such as copy, move, delete, and check-in) to be configured for control by a workflow. For example, the `FileManager.delete()` operation will invoke a DeleteRequest if workflow is enabled on the instance, and the Delete operation is workflow-enabled for the item being deleted.

The RequestManager supports the following operations:

- retrieve a list of requests submitted by the given user matching the given request criteria
- retrieve a list of requests for which the given user is responder to matching the given request criteria
- delete one or more requests with given IDs
- approve request with given ID for the given user
- deny request with the given ID for the given user
- cancel request with the given ID for the given user
- acknowledge request with the given ID for the given user
- invoke a "UserRequest" for the given set of target items. If the target item is controlled by a UserRequestWorkflowConfiguration, the associated custom workflow will be activated.
- return whether the given operation/action is workflow-enabled (i.e. request based) for the given item

ServiceToServiceManager

The ServiceToServiceManager uses the Service to Service (S2S) authentication framework to allow a trusted partner application to establish user sessions with a trusting provider application on behalf of its users, without having to supply any credentials for the users. The partner application instead supplies with each user session login request a digest credential (or potentially basic credential over HTTPS) that is used to authenticate the partner as being trusted to the provider service.

Note: The AXIS Java client stubs shipped with Oracle Content Services do not support digest authentication (see *http://ws.apache.org/axis/java/security.html*). Clients that use the Java stubs can authenticate through basic authentication, or by using an HTTP client and the S2S servlet.

Oracle Content Services 10g operates as the trusting provider service, with the partner service being potentially any application (registered/configured with OID) that is capable of establishing a client SOAP over HTTP Web Service connection with digest authentication headers. For this manager to function, the server must be configured for S2S authentication and the instance property `IFS.DOMAIN.CREDENTIALMANAGER.ServiceToServiceAuthenticationEnabled` set to true.

The ServiceToServiceManager provides the ability to:

- Authenticate a user with given username. The S2S header ORA_S2S_PROXY_USER must also be set to a non-null value corresponding to a valid OID user (it can be any user, such as `orcladmin`)

As an example, consider a custom portlet within Oracle Portal that displays a user's most recently accessed documents in Oracle Content Services. The sequence of operations would be similar to the following:

1. User "matt" authenticates to Oracle Portal (through the Oracle Login Server)
2. A default portal user home page containing the "most recent documents" portlet is requested to be displayed.
3. The portlet checks to see if an existing Oracle Content Services session cookie exists for the user in the user’s HttpSession (or equivalent) object store. If no existing session exists, proceed with service to service authentication (step 4). Otherwise, obtain a FileManager (step 7) and make the appropriate call to retrieve the documents.
4. The portlet obtains the portal service's credentials from a credential store (such as Oracle Internet Directory).
5. The portlet initiates a Web Service request to the Oracle Content Services ServiceToServiceLoginManager with digest authentication HTTP headers present that identify the partner service (Oracle Portal), and HTTP header `ORA_S2S_PROXY_USER` set to a non-null value (for example, "matt"). The ServiceToServiceLoginManager login method is called, supplying the user name of the portal authenticated user ("matt" [@non-default realm]).
6. A session cookie is returned that identifies the newly created Oracle Content Services user session, which is then stored in an appropriate location, such as the user’s HttpSession object.
7. Using the session cookie, obtain a FileManager, and then call the `getMostRecentDocuments()` method.
8. Process the returned item array and render portlet results.

A
Oracle Content Services Roles

Table A–1

Administrative Role	**Permissions**	**Applicable to Domain (D), Container (C), or Workspace (W)**			**Propagating?**
CategoryAdministrator	Discover, AdministerCategory	D			false
ConfigurationAdministrator	Discover, AdministerConfiguration	D	C	W	true
ContainerAdministrator	Discover, AdministerContainer, CreateContainer	D	C		true
ContentAdministrator	Discover, AddItem, AddVersion, Copy, CreateFolder, Delete, GetContent, GetMetadata, Lock, Move, SetAttribute, SetContent, SetMetadata	D	C	W	true
DomainAdministrator	Discover, AdministerDomain	D			false
QuotaAdministrator	Discover, AdministerQuota	D	C		true
RecordsAdministrator	Discover, AdministerRecord	D			false
RoleAdministrator	Discover, AdministerRole	D			false
SecurityAdministrator	Discover, AdministerSecurity	D	C	W	true
UserAdministrator	Discover, AdministerUser	D			false
WorkspaceAdministrator	Discover, AdministerWorkspace, CreateWorkspace	D	C	W	true

Table A–2

Standard (Non-Administrative) Role	**Permissions**	**Applicable to Domain (D), Container (C), or Workspace (W)**			**Propagating?**
Administrative Assistant	Discover, AddItem, AdministerConfiguration, AdministerSecurity, CreateFolder			W	false
Administrator	Discover, AddItem, AddVersion, AdministerConfiguration, AdministerSecurity, AdministerWorkspace, Copy, CreateFolder, Delete, GetContent, GetMetadata, Lock, Move, SetAttribute, SetContent, SetMetadata			W	true
Approver	Discover, Copy, GetContent, GetMetadata, Lock, SetAttribute, SetContent, SetMetadata			W	false
Author	Discover, AddItem, AddVersion, Copy, CreateFolder, Delete, GetContent, GetMetadata, Lock, Move, SetAttribute, SetContent, SetMetadata			W	false
Commentator	Discover, Copy, GetContent, GetMetadata, Lock, SetAttribute, SetContent, SetMetadata			W	false
ContainerViewer	Discover	D	C		false
ContentEditor	Discover, AddItem, AddVersion, Copy, CreateFolder, GetContent, GetMetadata, Lock, SetAttribute, SetContent, SetMetadata			W	false
Custodian	Discover, AddItem, AddVersion, Copy, CreateFolder, GetContent, GetMetadata, Lock, SetAttribute, SetContent, SetMetadata			W	false
Discoverer	Discover			W	false
LimitedAuthor	Discover, AddItem, AddVersion, Copy, CreateFolder, GetContent, GetMetadata, Lock, SetAttribute, SetContent, SetMetadata			W	false
Manager	AdministerSecurity, CreateFolder			W	false
None	NONE				false
Organizer	Discover, Copy, Delete, GetMetadata, Lock, Move, SetAttribute, SetMetadata			W	false
Participant	Discover, AddItem, AddVersion, Copy, CreateFolder, Delete, GetContent, GetMetadata, Lock, Move, SetAttribute, SetContent, SetMetadata			W	false

Table A–2 (Cont.)

Standard (Non-Administrative) Role	Permissions	Applicable to Domain (D), Container (C), or Workspace (W)			Propagating?
Reader	Discover, Copy, GetContent, GetMetadata			W	false
Reviewer	Discover, Copy, GetContent, GetMetadata			W	false
WorkspaceCreator	Discover, CreateWorkspace	D	C		true

Table A–3 FDK Constants for defined Roles

Role	FdkConstant
CategoryAdministrator	`ECM_ROLEKEY_CATEGORYADMIN`
ConfigurationAdministrator	`ECM_ROLEKEY_CONFIGURATIONADMIN`
ContainerAdministrator	`ECM_ROLEKEY_CONTAINERADMIN`
ContentAdministrator	`ECM_ROLEKEY_CONTENTADMIN`
DomainAdministrator	`ECM_ROLEKEY_DOMAINADMIN`
QuotaAdministrator	`ECM_ROLEKEY_QUOTAADMIN`
RecordsAdministrator	`ECM_ROLEKEY_RECORDSADMIN`
RoleAdministrator	`ECM_ROLEKEY_ROLEADMIN`
SecurityAdministrator	`ECM_ROLEKEY_SECURITYADMIN`
UserAdministrator	`ECM_ROLEKEY_USERADMIN`
WorkspaceAdministrator	`ECM_ROLEKEY_WORKSPACEADMIN`
ContainerViewer	`ECM_ROLEKEY_CONTAINERVIEWER`
WorkspaceCreator	`ECM_ROLEKEY_WORKSPACECREATOR`
Administrator	`ECM_ROLEKEY_ADMIN`

Table A–4 Permissions for Oracle Content Services roles

Permission	Description	FDK Constant
AddItem	Add an item to a folder (by create, or move operation)	`CAPABILITY_ADDITEM`
AddVersion	Add a new version to a version controlled document item	`CAPABILITY_ADDVERSION`
AdministerConfiguration	Create, modify, or delete configuration categories on an item (with the exception of SecurityConfiguration and QuotaConfiguration)	`CAPABILITY_ADMINISTER_ CONFIGURATION`
AdministerContainer	Modify or delete a container. Permission is required on the parent item of the container being modified or deleted.	`CAPABILITY_ADMINISTER_CONTAINER`

Table A–4 (Cont.) Permissions for Oracle Content Services roles

Permission	Description	FDK Constant
AdministerCategory	Create, modify, or delete a category class object	`CAPABILITY_ADMINISTER_CATEGORY`
AdministerDomain	Modify a domain's properties	`CAPABILITY_ADMINISTER_DOMAIN`
AdministerQuota	Modify the quota configuration of a workspace item	`CAPABILITY_ADMINISTER_QUOTA`
AdministerRecord	Create, modify, or delete a record file plan. Also allows user to remove "record" status from an existing record item and perform other records management administration.	`CAPABILITY_ADMINISTER_RECORD`
AdministerRole	Create, modify, or delete a custom role	`CAPABILITY_ADMINISTER_ROLE`
AdministerSecurity	Create, modify, or delete security configuration of an item	`CAPABILITY_ADMINISTER_SECURITY`
AdministerUser	Modify or delete a domain's users and groups. Additionally, enables user to get and set user preferences including domain defaults.	`CAPABILITY_ADMINISTER_USER`
AdministerWorkspace	Modify or delete a workspace	`CAPABILITY_ADMINISTER_WORKSPACE`
Copy	Copy an item	`CAPABILITY_COPY`
CreateContainer	Create a container	`CAPABILITY_CREATECONTAINER`
CreateFolder	Create a folder	`CAPABILITY_CREATEFOLDER`
CreateWorkspace	Create a workspace (not needed for creation of a personal workspace)	`CAPABILITY_CREATEWORKSPACE`
Delete	Delete an item	`CAPABILITY_DELETE`
Discover	Discover an item and view its basic metadata (such as name, description, and creation date). Permission is implicit if the user is granted any other permission on the item.	`CAPABILITY_DISCOVER`
GetContent	Get the content of a document item	`CAPABILITY_GET_CONTENT`
GetMetadata	Get the metadata (category information) of an item	`CAPABILITY_GETMETADATA`
Lock	Lock a document item	`CAPABILITY_LOCK`
Move	Move an item. Requires AddItem permission on the destination folder.	`CAPABILITY_MOVE`
SetAttribute	Set basic attributes of an item (description). Permission is required to rename Document, Folder, Family, and Link items. For link items, this permission also allows users to change the object referenced by the link.	`CAPABILITY_SET_ATTR`

Table A–4 (Cont.) Permissions for Oracle Content Services roles

Permission	Description	FDK Constant
SetContent	Set the content of a non-version-controlled document item	`CAPABILITY_SET_CONTENT`
SetMetadata	Set the metadata (create, modify, or delete category information) of an item	`CAPABILITY_SETMETADATA`

Notes on permission types:

- Within a single grant, the same Role may not appear more than once
- Within a single grant, the "NONE" Role may not be combined with any other role
- If the grantee belongs to the "World" group, the Domain must be enabled for world group grants.
- The SetAttribute permission is required to rename a Document, Folder, Family, or Link.
- The AdministerWorkspace permission is required to rename a Workspace.
- The AdministerContainer permission is required to rename a Container.
- The System Admin privileges are required to rename a Domain.
- The SecurityAdministrator role is the most powerful; users granted this role can grant themselves or anybody else all available access.
- To delete a Container, a user must have the AdministerContainer permission on the parent of the container being deleted.
- When deleting a Container, all recursively contained sub-containers and sub-workspaces are also deleted. The user must have permission to delete the sub-containers according to the rule stated above. The user must also have AdministerWorkspace permission on all of the sub-workspaces that are to be deleted. If the user does not have these required permissions, the originating container delete will fail with an `ACCESS_DENIED` exception. Containers that are deleted are permanently deleted; deleted workspaces have the workspace's contents moved to the archive.

Index