WHITEPAPER

Flatirons Solutions
Vision. Experience. Engineering Excellence.

Two Approaches for Migrating Existing Documentum Users, Groups, ACLs and Registered Tables

M. Scott Roth
And
Brian Yasaki

January 6, 2009

Table of Contents

Introduction

In Scott's previous article, *Content Migration Approaches for Documentum*, he discussed several content migration strategies that can be employed to migrate content from one Documentum repository to another. Each strategy had its advantages and disadvantages, and appropriate operational scenarios. However, none of these approaches specifically addressed a fundamental prerequisite requirement for successfully and completely migrating content: users, groups, ACLs and sometimes registered tables also need to be migrated.

In this article we specifically address how to migrate existing users, groups, ACLs and registered tables so that regardless of how the content is migrated, you are not faced with the administrative nightmare of reassign content to users after the fact. Most migration tools and techniques do not migrate users, groups and ACLs[1], and none migrate the content of registered tables. However, we have developed two simple solutions[2] for migrating existing user, groups, ACLs – and even the contents of registered tables.

The first approach uses Documentum's built-in Dump and Load operations. Usually, Dump and Load get a bad rap when discussing content migration because of their many shortcomings. However, Dump and Load are excellent solutions for migrating users, groups and ACLs. Dump and Load will even migrate your registered table objects (`dm_registered`), but not the actual content of the tables.

The second approach is to write a little DFC code to generate API scripts that recreate users, groups and ACLs in the target repository. This approach is not so much a migration as you usually think of one, but more of a recreation. We sort of think of this approach as being like the transporter in *Star Trek*. Users are disassembled from their current location (the source repository), stored in a buffer (the API script), and then reconstituted in an entirely new environment (the target repository). Using this approach, you can also migrate the *content* of registered tables; we'll show you how later in this article.

To set these two approaches into proper context, let us explain the situation we faced when we developed them. A customer had an existing Documentum environment with five years of content on a Windows/SQL Server platform. They wanted to decommission this environment and re-establish their Documentum repositories in a Unix/Oracle environment. To migrate the content we used a commercial migration tool, but before we could do that, we had to establish the user base in the Unix/Oracle environment. Thus these two approaches emerged.

[1] It's important to note that Documentum's Application Builder does not let you include users, groups and ACLs in a DocApp. Some commercial applications such as, Crown Partners' Buldoser, Generis' DocMigrator/SysMigrator, and Impact Systems' Q-Config all migrate users, groups and/or ACLs to some degree.
[2] These solutions were developed and tested on Documentum 4i and 5.3 SP5.

WHITEPAPER

Dump and Load

Using Dump and Load to migrate users, groups and ACLs turns out to be a really straightforward approach and is, in fact, how Documentum "migrates" these objects among members of a federated repository system.

Dump

The Dump operation is initiated by creating and saving a Dump object (`dm_dump_record`) in the repository. We usually create a small script in NotePad to create this object, set the requisite attributes, and save the object, and then run it in the IAPI32 command line utility. The following script is representative of a script that will dump all users, groups, ACLs and registered table objects. There are many options available for the Dump operation, consult the *Content Server Administrator's Guide* for more details.

```
create,c,dm_dump_record

# fully qualified file name for the dump file
set,c,l,file_name
c:\temp\users_groups_acls_regtbls.dmp

# set the user type to be dumped
append,c,l,type
dm_user

# dump all users by supplying a predicate that is always true
append,c,l,predicate
1=1

# set the group type to be dumped
append,c,l,type
dm_group

# dump all groups by supplying a predicate that is always true
append,c,l,predicate
1=1

# set the acl type to be dumped
append,c,l,type
dm_acl

# dump all acls by supplying a predicate that is always true
append,c,l,predicate
1=1

# set the registered table type to be dumped
append,c,l,type
dm_registered

# dump all registered tables by supplying a predicate that is
# always true
append,c,l,predicate
1=1
```

```
# save the object – this starts the dump
save,c,l

# check for dump errors
getmessage,c
```

Run this script using the IAPI32 command line utility or from the API editor in the Documentum Administrator on the source repository.

Load

Move the dump file (`c:\temp\users_groups_acls_regtbls.dmp`) to the target repository and Load it. Loading dump files is also performed by writing a small API script. Like the Dump operation, the Load operation is initiated by creating and saving a Load object (`dm_load_record`) in the repository. Again, there are many options available for the Load operation, consult the *Content Server Administrator's Guide* for more details.

```
create,c,dm_load_record

# fully qualified dump file name to load
set,c,l,file_name
c:\temp\users_groups_acls_regtbls.dmp

# save the object – this starts the load
save,c,l

# check for load errors
getmessage,c
```

You have now completely migrated all of the users, groups, ACLs and registered table objects from the source repository to the target repository—and it was pretty easy. There are only two small disadvantages to this approach and depending upon your migration scenario, they might not even apply.

1. You have no opportunity to review or modify the users, groups, ACLs or registered tables that are being migrated except through the Documentum Administrator before the Dump operation is initiated. If a problem occurs with the Dump or the Load operation, you have no visibility into the dump file to troubleshoot the problem. You also have no opportunity to do any transformations or editing of the objects being migrated.
2. The registered table objects (`dm_registered`) were migrated, but the underlying tables they represent were not. You still need to figure out how to migrate the content of those tables to the target system. Depending upon your database vendor, you probably have tools to export/dump/archive these tables, move them to the target system, and

import/load/restore them. In our situation, this problem was exaggerated because we were dealing with databases from two different vendors.

DFC and API Scripts

This approach to migrating users, groups, ACLs and registered tables requires writing a little DFC code that in turn writes some API scripts to facilitate the migration. This approach is infinitely more fun than the Dump and Load approach, but depending upon your situation, "fun" might not be what you're after. Here is the basic premise of this approach:

- Create a simple Java application that logs in and establishes a session to the source repository.
- Run some queries to gather all the users, groups, ACLs and registered table objects that you are interested in migrating. One advantage to this approach is that you can make the queries as selective as necessary.
- As you iterate through each collection returned by the queries, extract the necessary metadata to recreate the objects and write API commands to a text file to facilitate their recreation. Another advantage to this approach is the opportunity to tweak any of the metadata either before you write it to the output file or editing the metadata while reviewing the text file.
- Process the rows of each registered table and write a DQL script to recreate the registered table.
- Transport these scripts to the target environment and execute them using IAPI32 and IDQL32.

One very important lesson we learned from implementing this approach is that users, groups and ACLs can be a tangled mess. Frequently you encounter a chicken and egg sort of paradox: you can't create an ACL because a group doesn't exist, but you can't create the group because the user doesn't exist and you can't create the user because their default ACL doesn't exist.

We discovered that the solution to this problem is to create these objects in several iterative passes. The order we determined to work best is this:

1. Groups – create the basic group objects with proper names but no members.
2. Users – create user objects with no default groups or ACLs.
3. ACLs – create basic ACL objects with groups and/or users.
4. Patch Groups – now that the users exist, add them to the groups
5. Patch Users – now that all the ACLs and groups exist, update the user objects.

Following are representative examples of the code to create the five API scripts listed above.

Groups

This code simply queries for all of the `dm_group` objects and writes two output files. You can augment this query to target or eliminate certain users if necessary. The first output file creates the API script to create the groups with minimal metadata and no membership (script 1). You could obviously collect more metadata if you need additional metadata about each group. The second API file writes the "patch" (script 4) that will be used to add membership to each group once the user objects have been migrated.

```
private void exportGroups(IDfSession s) throws DfException {
    outputFile1 = openOutputFile("1_groups.api");
    outputFile2 = openOutputFile("4_groups.api");

    q.setDQL("select r_object_id from dm_group");
    col = q.execute(s, DfQuery.DF_READ_QUERY);

    while (col.next()) {

        IDfGroup group = (IDfGroup) s.getObject(new DfId(
          col.getString("r_object_id")));

        // Create group objects

        outputFile1.println("create,c,dm_group");
        outputFile1.println("set,c,l,group_name");
        outputFile1.println(group.getGroupName());

        if (group.getDescription().length() > 0) {
            outputFile1.println("set,c,l,description");
            outputFile1.println(group.getDescription());
        }

        // Repeat this pattern for additional attributes

        outputFile1.println("save,c,l");

        // Write the file that will later patch the membership

        outputFile2.println("retrieve,c,dm_group where group_name = '"
          + group.getGroupName()+ "'");
        outputFile2.println("set,c,l,owner_name");
        outputFile2.println(group.getOwnerName());

       // Add member groups

        if (group.getGroupsNamesCount() > 0) {
            int cnt = group.getGroupsNamesCount();
            for (int i=0; i<cnt; i++) {
                outputFile2.println("append,c,l,groups_names");
                outputFile2.println(group.getGroupsNames(i));
            }
        }
        // Add member users

        if (group.getAllUsersNamesCount() > 0) {
            int cnt = group.getAllUsersNamesCount();
            for (int i=0; i<cnt; i++) {
                outputFile2.println("append,c,l,users_names");
                outputFile2.println(group.getUsersNames(i));
            }
        }
        outputFile2.println("save,c,l");
    }

    // Close everything

    col.close();
    closeOutputFile(outputFile1);
    closeOutputFile(outputFile2);
}
```

The result of this pseudo code will be two API files similar to these.

```
### 1_groups.api ###
create,c,dm_group
set,c,l,group_name
mac support
set,c,l,description
support for mac users
save,c,l

create,c,dm_group
set,c,l,group_name
pc support
set,c,l,description
support for pc users
save,c,l

### 4_groups.api ###
retrieve,c,dm_group where group_name = 'mac support'
append,c,l,groups_names
admingroup
append,c,l,users_names
Steve Jobs
append,c,l,users_names
Steve Wozniak
save,c,l

retrieve,c,dm_group where group_name = 'pc support'
append,c,l,groups_names
admingroup
append,c,l,users_names
Bill Gates
append,c,l,users_names
Paul Allen
save,c,l
```

Users

Like the Group code, the logic to migrate the Users queries for all users (`dm_user`) in the repository. You could restrict this query to only process active users or some other subset if necessary. Also like the Group code, this code writes two output files. The first output file (script 2) creates the basic user objects with the some basic metadata. The second file (script 5) "patches" the user objects with their default ACLs after the ACLs are migrated.

```
private void exportUsers(IDfSession s) throws DfException {
    outputFile1 = openOutputFile("2_users.api");
    outputFile2 = openOutputFile("5_users.api");

    q.setDQL("select r_object_id from dm_user where r_is_group =
      FALSE");
    col = q.execute(s, DfQuery.DF_READ_QUERY);

    while (col.next()) {

        IDfUser user = (IDfUser) s.getObject(new DfId
          (col.getString("r_object_id")));

            // Create user objects

            outputFile1.println("create,c,dm_user");
            outputFile1.println("set,c,l,user_name");
            outputFile1.println(user.getUserName());

            outputFile1.println("set,c,l,client_capability");
            outputFile1.println(user.getClientCapability());

            outputFile1.println("set,c,l,default_folder");
            outputFile1.println(user.getDefaultFolder());

            if (user.getUserGroupName().length() > 0) {
                outputFile1.println("set,c,l,user_group_name");
                outputFile1.println(user.getUserGroupName());
            }

            if (user.getUserOSName().length() > 0)  {
                outputFile1.println("set,c,l,user_os_name");
                outputFile1.println(user.getUserOSName());
            }

            outputFile1.println("set,c,l,user_privileges");
            outputFile1.println(user.getUserPrivileges());

            outputFile1.println("set,c,l,user_state");
            outputFile1.println(user.getUserState());

            // Repeat pattern for additional attributes

            outputFile1.println("save,c,l");

            // Write the file that will later patch the users
            outputFile2.println("retrieve,c,dm_user where user_name =
              '" + user.getUserName() + "'");

            outputFile2.println("set,c,l,acl_name");
            outputFile2.println(user.getACLName());

            outputFile2.println("set,c,l,acl_domain");
            outputFile2.println(user.getACLDomain());

            outputFile2.println("save,c,l");
    }

    // Close everything

    col.close();
    closeOutputFile(outputFile1);
    closeOutputFile(outputFile2);
}
```

The result of this pseudo code will be two API files similar to these.

```
### 2_users.api ###
create,c,dm_user
set,c,l,user_name
bgates
set,c,l,client_capability
8
set,c,l,default_folder
/PC
set,c,l,user_group_name
pc support
set,c,l,user_os_name
bgates
set,c,l,user_privileges
16
set,c,l,user_state
0
save,c,l

create,c,dm_user
set,c,l,user_name
sjobs
set,c,l,client_capability
8
set,c,l,default_folder
/MAC
set,c,l,user_group_name
mac support
set,c,l,user_os_name
sjobs
set,c,l,user_privileges
16
set,c,l,user_state
0
```

```
save,c,l

### 5_users.api ###
retrieve,c,dm_user where user_name = 'Bill Gates'
set,c,l,acl_name
pc projects acl
set,c,l,acl_domain
dmadmin
save,c,l

retrieve,c,dm_user where user_name = 'Steve Jobs'
set,c,l,acl_name
mac projects acl
set,c,l,acl_domain
dmadmin
save,c,l
```

ACLS

The ACL code migrates all of the custom ACLs in the repository. This is accomplished by appending the `where object_name not like 'dm_%'` clause to the end of the DQL statement. Again, this DQL can be as restrictive or inclusive as needed—this is part of the advantage to this approach. The one important thing to remember about creating ACLs is that access must be granted; you can't set the attributes that contain the user and group access rights.

```
private void exportACLs(IDfSession s) throws DfException {
    outputFile = openOutputFile("3_acls.api");

    q.setDQL("select r_object_id from dm_acl where object_name not like
      'dm_%'");
    col = q.execute(s, DfQuery.DF_READ_QUERY);

    while (col.next()) {

        IDfACL acl = (IDfACL) s.getObject(new DfId
          (col.getString("r_object_id")));

        // Create ACL object

        outputFile.println("create,c,dm_acl");
        outputFile.println("set,c,l,object_name");
        outputFile.println(acl.getObjectName());

        if (acl.getDescription().length() > 0) {
            outputFile.println("set,c,l,description");
            outputFile.println(acl.getDescription());
        }

        outputFile.println("set,c,l,acl_class");
        outputFile.println(acl.getACLClass());

        outputFile.println("set,c,l,owner_name");
        outputFile.println(acl.getDomain());
       // Remember access has to be GRANTED

        if (acl.getAccessorCount() > 0) {
            for (int i=0; i< acl.getAccessorCount(); i++) {
                outputFile.println("grant,c,l,'" +
                  acl.getAccessorName(i) + "'," +
                  acl.getAccessorPermit(i));
            }
        }

       // Remember to handle extended permits too

        outputFile.println("save,c,l");
    }

    // Close everything

    col.close();
    closeOutputFile(outputFile);
}
```

The result of this pseudo code will be an API file similar to this.

```
### 3_acls.api ###
create,c,dm_acl
set,c,l,object_name
pc_support
set,c,l,description
PC Support ACL
set,c,l,acl_class
0
set,c,l,owner_name
dmadmin
grant,c,l,'dm_world',2
grant,c,l,'dm_owner',7
grant,c,l,'admingroup',7
grant,c,l,'pc support',7
save,c,l

create,c,dm_acl
set,c,l,object_name
mac_support
set,c,l,description
MAC Support ACL
set,c,l,acl_class
0
set,c,l,owner_name
dmadmin
grant,c,l,'dm_world',2
grant,c,l,'dm_owner',7
grant,c,l,'admingroup',7
grant,c,l,'mac support',7
save,c,l
```

Running these API scripts in their numbered order on the target repository will recreate the users, groups and ACLs that are in your source repository.

- `1_groups.api`
- `2_users.api`
- `3_acls.api`
- `4_groups.api`
- `5_users.api`

Registered Tables

Finally, let's look at migrating registered tables. This approach works in two steps. In the first step, each table is examined and converted to DQL. In the second step, each registered table object (`dm_registered`) is captured for migration.

The *createRegTableScript()* method described below interrogates each registered table and writes DQL to recreate it. First it writes the create table statement that specifies each columns' name, data type and length. Then it steps through each row and writes insert statements for each value it finds. Notice that each statement the script produces is prefaced with "`execute exec_sql with query =`". This allows the script to be run using the IDQL32 command line tool and eliminates the need to create the tables using a database-specific utility.

```
private void createRegTableScript(IDfSession s) throws DfException {
    String[] dataTypes = null;
    outputFile = openOutputFile("regtab_script.dql");

    // Determine database type

    if (s.getDBMSName().equalsIgnoreCase("oracle")) {
         dataTypes = ORCLdataTypes;
    } else {
         dataTypes = SQLdataTypes;
    }

    // Get custom registered tables

    q.setDQL("select r_object_id from dm_registered where object_name
      not like 'dm%' and 'adm_turbo%' and object_name not like '%_s'
      and object_name not like '%_r'");
    col = q.execute(s, DfQuery.DF_READ_QUERY);

    while (col.next()) {

        StringBuilder colSpecs = new StringBuilder();
        StringBuilder colNames = new StringBuilder();

        IDfSysObject regtab = (IDfSysObject) m_session.getObject(new
          DfId(col.getString("r_object_id")));
        // Write table drop
        outputFile.println("execute exec_sql with query = 'drop table
          dbo." + regtab.getString("object_name") + "'");
        outputFile.println("go");

        // Query each table
        q2.setDQL("select * from dbo." + regtab.getString
          ("object_name"));
        col2 = q2.execute(s,DfQuery.DF_READ_QUERY);

        // Write column info
        IDfTypedObject colRow = col2;

        // Build strings with column specs (for create) and column
        // names (for insert)
        for (int i=0; i<colRow.getAttrCount(); i++) {

            if (colNames.length() > 0) {
                colNames.append(",");
            }

            // Create col names
            colNames.append(colRow.getAttr(i).getName());

            if (colSpecs.length() > 0) {
                colSpecs.append(",");
            }

            // Create the insert info
            colSpecs.append(colRow.getAttr(i).getName() + " " +
              dataTypes[colRow.getAttr(i).getDataType()]);

            // If it's a string, get size
            if (colRow.getAttr(i).getDataType() == 2) {
                colSpecs.append("(" + colRow.getAttr(i).getLength() +
                  ")");
            }
        }

        // Create table
        outputFile.println("execute exec_sql with query = 'create table
          dbo." + regtab.getString("object_name") + " (" +
          colSpecs.toString() + ")'");
        outputFile.println("go");

        // Create insert statements
        while (col2.next()) {
            outputFile.print("execute exec_sql with query = 'insert
              into dbo." + regtab.getString("object_name") + "(" +
              colNames.toString() + ") values (");

            StringBuilder values = new StringBuilder();
            colRow = col2;

            // Get value in each row
            for (int i=0; i<colRow.getAttrCount(); i++) {
                if (values.length() > 0) {
                    values.append(",");
                }

                // Get values and escape single ticks
                String row = colRow.getString(colRow.getAttr(i).
                  getName());
                row.replaceAll("'","\'");
                values.append("''" + row + "''");
            }

            outputFile.println(values.toString() + ")'");
            outputFile.println("go");
        }

        // Close the inner collection
        col2.close();
        outputFile.println();

    }

    // Close outer collection
    col.close();
    closeOutputFile(outputFile);
}
```

The result of this pseudo code will be a DQL file similar to this.

```
### regtab_script.dql ###
execute exec_sql with query = 'drop table dm_dbo.states'
go
execute exec_sql with query = 'create table dm_dbo.states (name
    STRING(32), abbr STRING(2))'
go
execute exec_sql with query = 'insert into dm_dbo.states (name,abbr)
    values (''Virginia'', ''VA'')'
go
execute exec_sql with query = 'insert into dm_dbo.states (name,abbr)
    values (''Maryland'', ''MD'')'
execute exec_sql with query = 'insert into dm_dbo.states (name,abbr)
    values (''District of Columbia'', ''DC'')'
```

The second step of the process is to create the registered table (`dm_registered`) objects for each of the tables captured in step 1. This is a straightforward process and capitalizes on the fact that Documentum does not really need the name, size and length of each column in the underlying table; it will allow you to simply get away with using a dummy column name.

```
private void exportRegTables(IDfSession s) throws DfException {
    outputFile = openOutputFile("dmregistered_script.dql");

    // create RDBMS tables first
    createRegTableScripts();
    q.setDQL("select * from dm_registered where object_name not like
      'dm_%'");
    col = q.execute(s, DfQuery.DF_READ_QUERY);

    while (col.next()) {

        // get object
        IDfSysObject regtab = (IDfSysObject) s.getObject(new
          DfId(col.getString("r_object_id")));
        outputFile.println("register table dm_dbo." +
          regtab.getString("table_name") + " (dummy string(10))" );
        outputFile.println("go");
    }

    // Close everything
    col.close();
    closeOutputFile(outputFile);
}
```

The result of this pseudo code will be a DQL file similar to this.

```
### dm_registered_script.dql ###
register table dm_dbo.states (dummy string(10))
go
```

Now, take these scripts to the target system and reconstitute your registered tables by running first the regtab_script.dql and then the dmregistered_script.dql using the IDQL32 command line tool.

There you have it; you have completely migrated the users, groups, ACLs and registered tables from your source repository to your target repository. This approach has some obvious advantages over the Dump and Load method:

1. You can easily control the objects that are migrated by modifying the DQL statements used. This gives you infinite flexibility to control batch sizes, or create migration scripts based upon some other attribute of the user.
2. You can use conditional logic while processing the collection that results from the query. You can transform metadata, map metadata to new or different attributes, consolidate user groups, etc. This injects a degree of "ETL-ness" into your user migration.
3. You can examine the objects that are being migrated. Once the API scripts are written, you can review the objects that are being migrated and make adjustments if necessary. This provides a degree of QA that the Dump and Load method does not.
4. Registered tables—both objects and content—can also be migrated using only DQL; there is no need for any special database tools. In fact, this entire approach is database and repository version neutral. You run these

export scripts on a Windows/SQL Server/5.3 system and import the users, groups, ACLs and registered tables into a Unix/Oracle/D6 target system.

One final advantage: if you are clever, you can have the Java code write an "undo" script for you also in case you need to remove the objects from the target repository.

Conclusion

In this article we have given you two quick approaches for migrating existing users, groups, ACLs and registered tables from a source repository to a target repository. The first method utilized Dump and Load and is really simple to implement. The simplicity comes at the cost of flexibility. If you are migrating a simple repository and don't necessarily need to remap attributes or do anything else to the objects as they migrate, this is a good approach. This approach sticks to proven industry tools (Documentum's Dump and Load, and your database vendor's export/import tools) and requires no real programming.

The second approach used a Java/DFC program to extract user, group, ACL and registered table objects from the repository and persisted them as a Documentum API script file. This approach we equated to using Star Trek's transporter: you extracted objects, persisted them on a medium, and reconstituted them in a new place. This approach required an investment in writing some code, but gave you maximum control and flexibility in your migration. While extracting objects from the source repository, you could do anything imaginable to them before writing them to the API file. This approach will even allowed you to extract registered table content. The key to this approach was the specific order in which objects are reconstituted and patched in the target repository.

There are many approaches and philosophies for migrating Documentum content; these are but two. We're certain you have encountered or thought about other objects that need migration that these approaches don't address (e.g., audit trails, version stacks). Those situations, requirements and experiences can be the subject of your article. We look forward to reading it!

About Flatirons Solutions

Flatirons Solutions, an Inc. 500 company, provides consulting, systems integration, and systems & software engineering services to Fortune 500 companies and government agencies. A leading content management solutions provider specializing in XML-based publishing and digital asset management, Flatirons has provided enterprise-wide solutions in industries such as high technology, aerospace, transportation, publishing, manufacturing, financial services, insurance, media and entertainment, retail, and healthcare. Flatirons Solutions also actively participates in both DITA and DocBook XML technical committees. Established in 2001, Flatirons Solutions is a privately-held company headquartered in Boulder, Colorado, with offices in Washington D.C. and Ft Worth, TX. For more information visit Flatirons Solutions on the web at http://www.FlatironsSolutions.com.

About the Authors

M. Scott Roth is the content management solutions architect for Government Solutions at Flatirons Solutions Corp. He is also the author of the book, A Beginners Guide to Developing Documentum Desktop Applications, and the open source Documentum command line client, DOCS.

<SDG><

Brian Yasaki is a content management consultant for Government Solutions at Flatirons Solutions Corp.

Flatirons Solutions
Vision. Experience. Engineering Excellence.

4747 Table Mesa Drive, Suite 200
Boulder, Colorado 80305
(303) 544-0514
info@FlatironsSolutions.com

www.FlatironsSolutions.com