An Introduction To

Db::Documentum

M. Scott Roth

September 2001

Version 1.2

Contents

Figures

Revisions

1.2	September 2001	Updated for Db::Documentum 1.5. Updated the idql.pl script. Minor editorial changes.
1.1	October 2000	Updated for Db::Documentum 1.4. Editorial improvements.
1.0	July 2000	Initial Release

1 Introduction

The Db::Documentum module is an interface between Perl and Documentum's Enterprise Document Management System (EDMS). It provides access to Documentum's API from Perl, thus allowing you to program for Documentum with Perl.

Documentum is an industry leader in enterprise document management systems. Its server contains an extensive API (~148 methods) that provides access to all areas of the EDMS. Perl is an excellent choice for programming in Documentum. It has all the capabilities of a sophisticated programming language (I/O, logic, extensibility--not to mention regular expressions) in addition to the freedom and flexibility you expect from a scripting language. The marriage of Perl and Documentum seems natural and exciting to me. I think you will agree.

This tutorial gives you an overview of the Db::Documentum module: how to install it, what it contains, and how to use it. It presents a number of real-world examples to demonstrate how easily you can program Documentum in Perl and how powerful the combination can be.

Note: This tutorial and its examples were written and tested on a PC running Microsoft Windows NT 4, Windows 2000 and ActiveState Perl 5.6 (build 616). Db::Documentum has been successfully run on other platforms and under other software configurations; see the documentation for details. Although nearly everything discussed here is applicable as-is, in UNIX there are a few exceptions. I will point them out along the way.

1.1 Requirements

To implement the examples discussed in this tutorial:

- You must have access to a working EDMS 98 or 4i Docbase. You will need sufficient privileges to create types, cabinets, and folders.
- You must have either the Documentum WorkSpace or 4i Desktop Client installed on your workstation.
- Depending on how you choose to install the Db::Documentum module, you may need a C/C++ compiler. If you choose the traditional method, you will need the compiler. If you choose the PPM method you will not.
- You will need the Db::Documentum module (version 1.5 or later).
- You will need a good working knowledge of Perl and Documentum's API.

1.2 Conventions

The following typographic conventions are used in this tutorial:

- `Courier Fixed Width` denotes Perl code, variable names, file names, OS commands, and other programmatic elements.
- *Italics* denotes Db::Documentum function names and URLs.
- *`Courier Fixed Width Italics`* denotes subroutine and Documentum method names.
- Arial denotes screen elements.

I use "`>`" to indicate the command line prompt.

1.3 Installation

If you are using Microsoft Windows NT, you have two options for installation: the traditional method or the PPM method. If you are using UNIX, use the traditional method.[1]

1.3.1 The Traditional Method

Download the module from the CPAN (*http://www.perl.com/CPAN-local/modules/by-module/Db/*), unpack it, and carefully read the `README` and the `Makefile.PL` files. `Makefile.PL` requires some tweaking to ensure the DMCL libraries and header files are in the proper locations on your hard drive and that those locations are communicated to `make`. `Makefile.PL` details all of the necessary files and paths. After tweaking, it's business as usual for the install:

```
>perl Makefile.PL
>nmake[2]
>nmake test
>nmake install
```

1.3.2 The PPM Method

If you are installing Db::Documentum on Windows NT and don't have a C/C++ compiler, you can download compiled versions of the module from: *http://www.erols.com/theroths/ perl_stuff.html*. After downloading and unpacking the archive, install it by typing:

```
>PPM install Db-Documentum.ppd
```

1.3.3 Installation Test

You can test your Db::Documentum installation by running the `test.pl` script in `/etc` of your installation directory. If you used the PPM method of installation, you will need to unpack

---

[1] If you don't have a C compiler on your UNIX box, you can download the pre-compiled module for Solaris 8 from: *http://www.erols.com/theroths/ perl_stuff.html*.

[2] In UNIX, this command is simply `make`.

the `Db-Documentum.tar.gz` file first.

1.4 Use

To use Db::Documentum and Db::Documentum::Tools, just `use` them in your Perl script.

```
#!/usr/local/bin/perl

use Db::Documentum qw (:all);
use Db::Documentum::Tools qw (:all);
.
.
.
```

Because the modules do not automatically import their functions into Perl's namespace, you must declare which functions in Db::Documentum and Db::Documentum::Tools you want to use. The keyword `:all` imports all of the functions. See the modules' source code for more details.

That's all there is to it!

2 Db::Documentum Module Overview

Documentum really has only three API functions: *dmAPIExec()*, *dmAPIGet()*, and *dmAPISet()*. These three provide access to all of the methods of the EDM Server and WorkSpace client. Through them, Perl interfaces with Documentum. This section discusses the three functions, what they do, and how to use them.

2.1 dmAPIExec()

The *dmAPIExec()* function executes EDM Server and Workspace methods. *dmAPIExec()* returns `TRUE` (1) or `FALSE` (0) based upon the success or failure of the method it executes.

2.1.1 Syntax

```
$api_stat = dmAPIExec("<method name>,<session id>,<method arguments>");
```

where `<method name>` is a Documentum method name, `<session id>` is a Documentum session identifier, and `<method arguments>` are arguments required by `<method name>`.[3]

2.1.2 Examples

```
dmAPIExec("close,c,$col_id");
```
[4]

closes the open collection identified by `$col_id`.

```
$api_stat = dmAPIExec("execquery,c,'F',select * from dm_document where owner_name =
user");
```

runs a DQL query against the Docbase. Remember, *dmAPIExec()* returns only `TRUE` or `FALSE`

[3] Consult the *Documentum Server Reference Manual* for a complete list of all methods and their arguments.

[4] You will often see the Documentum shorthand "c" used for the current session ID.

and not the result of the query. To obtain the result of the query, you must use a *dmAPIGet()* method as discussed below.

2.2 dmAPIGet()

The *dmAPIGet()* function retrieves information from the EDM Server. *dmAPIGet()* returns a scalar containing the information that was requested.

2.2.1 Syntax

```
$rv = dmAPIGet("<method name>,<session id>,<method arguments >");
```

where `<method name>` is a Documentum method name, `<session id>` is a Documentum session identifier, and `<method arguments>` are arguments required by `<method name>`.

2.2.2 Examples

```
$sessionID = dmAPIGet("connect,$docbase,$user,$password");
```

logs `$user` into `$docbase`, and returns `$sessionID`.

```
$last_col_id = dmAPIGet("getlastcoll,c");
```

returns the collection ID for the last executed query. Remember, the *dmAPIGet()* function returns a scalar, not `TRUE` or `FALSE`.

Using *dmAPIExec()* and *dmAPIGet()* you can query the Docbase and print the names of the documents you own.

```
# do query
$api_stat = dmAPIExec("execquery,c,'F',select * from dm_document where owner_name =
user");

# if query successful
if ($api_stat) {
    $col_id = dmAPIGet("getlastcoll,c");

            # if collection id obtained
            if ($col_id) {

                  # iterate over collection getting attrs and printing
                  while (dmAPIExec("next,c,$col_id")) {
                        $title = dmAPIGet("get,c,$col_id,title");
                        $obj_name = dmAPIGet("get,c,$col_id,object_name");
                        $owner = dmAPIGet("get,c,$col_id,owner_name");
                        print "$title ($obj_name) is owned by $owner\n";
                  }
            }

    # it is VERY important to close collections
    dmAPIExec("close,c,$col_id");
}
```

2.3 dmAPISet()

The *dmAPISet()* function sets the value of an attribute on an object. *dmAPISet()* returns `TRUE` or `FALSE` based upon the success or failure of setting the indicated value.

2.3.1 Syntax

```
$api_stat = dmAPISet("<method name>,<session id>,<method arguments >","<value>");
```

where `<method name>` is a Documentum method name, `<session id>` is a Documentum session identifier, `<method arguments>` are arguments required by `<method name>`, and `<value>` is the value being set by `<method name>`. *Note: unlike the other API functions, this function's signature contains two scalars separated by a comma.*

2.3.2 Examples

```
$api_stat = dmAPISet("set,c,$obj_id,title",$title);
```

sets the attribute `title` to the value contained in `$title` for the object identified by `$obj_id`.

```
$api_stat = dmAPISet("append,c,$obj_id,my_date,yyyymmdd","19890805");
```

appends '19890805' to the repeating attribute, `my_date`, of the object identified by `$obj_id` (using a custom date format).

Using *dmAPIExec()*, *dmAPIGet()*, and *dmAPISet()* you can search the Docbase for the documents you own and touch them.

```
# assume $now = today's date
# do query
$api_stat = dmAPIExec("execquery,c,'F',select * from dm_document where owner_name =
user");

# if query successful
if ($api_stat) {
     $col_id = dmAPIGet("getlastcoll,c");

     # if collection ID obtained
     if ($col_id) {

          # iterate over collection getting attrs, setting date_modified,
          # and printing
          while (dmAPIExec("next,c,$col_id")) {
                 $title = dmAPIGet("get,c,$col_id,title");
                 $obj_name = dmAPIGet("get,c,$col_id,object_name");
                 $obj_id = dmAPIGet("get,c,$col_id,r_object_id");
                 dmAPISet("set,c,$col_id,date_modified","$now,'dd/mm/yyyy'");

                 # remember to save the changes!
                 dmAPIExec("save,c,$obj_id");
                 print "$title ($obj_name) touched\n";
          }
     }
```

```
        # it is VERY important to close collections
        dmAPIExec("close,c,$col_id");
    }
```

2.4 dmAPIInit() and dmAPIDeInit()

Two other Documentum API functions are used in Db::Documentum: *dmAPIInit()* and *dmAPIDeInit()*. As their names imply, they initialize and deinitialize the API interface. These functions require no parameters and return `TRUE` or `FALSE` based upon their success or failure. The Db::Documentum module calls these functions automatically for you when your program begins and ends. You need not call them directly.

3 Db::Documentum::Tools Module Overview

Db::Documentum::Tools is a companion module to Db::Documentum. It contains subroutines for many common Documentum tasks.[5]. These tasks can be accomplished through the API--some easier than others--but encapsulating them in subroutines relieves you of unnecessary and tedious programming and helps to ensure consistent reuse.

This section discusses each of the module's subroutines, what they do, and how to use them. I encourage you to examine the code for the Db::Documentum::Tools module itself for additional insights.

3.1 dm_LastError()

dm_LastError() returns a scalar containing error messages for a particular session.

3.1.1 Syntax

```
$errors = dm_LastError(<session id>,<level>,<number>);
```

All of *dm_LastError()'s* arguments are optional. However, you will almost always call it with, at least, `<session id>`. The only exception is when you retrieve errors for a failed logon attempt and no `<session id>` exists. With no `<session id>` defined, *dm_LastError()* uses the `apisession` session ID. The `apisession` identifies the session created by *dmAPIInit()* when it initially connects to the server.

The arguments operate as follows:

> `<session id>` = the session ID of the messages to return. If not present, `apisession` assumed.
>
> `<level>` = the level of messages to return. This argument returns all messages equal to or less than the level setting. If not present, level 3 is assumed.
>
> 1 = Informational Messages

[5] There are additional subroutines in the Db::Documentum::Tools module that are not discussed here.

2 = Warning Messages

3 = Error Messages

4 = Fatal Error Messages.

<number> = the number of messages to return. The specified number of messages will be returned as a scalar delimited by "\n." If not present, all messages are returned.

3.1.2 Examples

```
$errors = dm_LastError();
```

returns all level 3 error messages for the `apisession` session (i.e., failed logon).

```
$info = dm_ LastError($sessionID,1);
```

returns all level 1 messages to `$info`.

```
print dm_ LastError($sessionID,3,1);
```

prints the last level 3 error message.

3.2 dm_Connect()

dm_Connect() logs a user onto the Docbase and returns the session ID if successful, or `undef` on failure.

3.2.1 Syntax

```
$sessionID = dm_Connect(<docbase>,<username>,<password>,<domain>,<user_arg>);
```

Depending upon which OS you use and how you authenticate users, the *dm_Connect()* syntax differs slightly. All OSs require the first three arguments: `<docbase>`, `<username>`, and `<password>`. `<domain>` is optional. If you use an authentication method other than that provided by Documentum, you can pass an additional argument, `<user_arg>`, to your authentication program. See the *Documentum Server Reference Manual* and the Db::Documentum::Tools source code for details and examples.[6]

3.2.2 Examples

```
# assume $DOCBASE, $USER, $PASSWD defined previously
$sessionID = dm_Connect($DOCBASE,$USER,$PASSWD);
die dm_LastError() unless $sessionID;
```

attempts to logon to `$DOCBASE` as `$USER` with password `$PASSWD` and returns the session ID

[6] The Db::Documentum::Tools module contains subroutines and information to assist users of Kerberos authentication.

to `$sessionID`. If no session ID is returned (i.e., logon failed), an error message is printed and the script `dies`.

3.3 dm_CreateType()

New object types are usually created in a Docbase using DQL because the Documentum API does not include a method to perform this task. Have no fear! Db::Documentum::Tools provides an easy-to-use subroutine for this job: *dm_CreateType()*.

3.3.1 Syntax

```
$api_stat = dm_CreateType(<object name>,<super type>,<attribute hash>);
```

where `<object name>` is any valid object name that you desire, `<super type>` is any Documentum object type that you can subtype, and `<attribute hash>` is an optional list of custom attributes. The `<attribute hash>` is a Perl hash where the attribute names are used as the hash keys, and the database field definitions as their values. The function can be called without a hash defined, in which case no additional attributes are added to the new object type. The *dm_CreateType()* returns `TRUE` or `FALSE` based upon its success or failure.

3.3.2 Examples

```
$api_stat = dm_CreateType ("my_document","dm_document");
```

creates an object type called `my_document` that is a subtype of `dm_document` with no customized attributes.

```
%ATTRS = (cat_id   =>  'CHAR(16)',
          locale   =>  'CHAR(255) REPEATING');
$api_stat = dm_CreateType ("your_document","my_document",%ATTRS);
```

creates an object type called `your_document` that is a subtype of `my_document` and contains two additional attributes: `cat_id` and `locale`.

3.4 dm_CreateObject()

dm_CreateObject() allows you to easily create a new instance of an existing object type, and optionally assign attribute values to it.

3.4.1 Syntax

```
$obj_id = dm_CreateObject (<object type>,<attribute hash>);
```

where `<object type>` is any valid object type that you can instantiate and `<attribute hash>` is an optional list of custom attributes and their values. The `<attribute hash>` is a Perl hash where the attribute names are used as the hash keys, and the attribute values as the hash values. In the case of repeating attributes, the hash values must be delimited by

`$Db::Documentum::Tools::Delimiter`. (The subroutine parses the hash values looking for this value. If it is found, the hash value is `split` on `$Db::Documentum::Tools::Delimiter` and *`appended()`* to the repeating attribute.) The function can also be called without the hash defined, in which case no attributes are set. *dm_CreateObject()* returns the newly created object's `r_object_id` upon its success, or `undef` on failure.

3.4.2 Examples

```
# get the repeating attr delimiter
$delim = $Db::Documentum::Tools::Delimiter;

%ATTRS = (object_name     => 'test_doc1',
          title           => 'My Test Doc 1',
          keywords        => 'Scott' . $delim . 'Test Doc' . $delim .
                             'Db-Documentum',
          r_version_label => 'TEST');

$obj_id = dm_CreateObject ("dm_document",%ATTRS);
$api_stat = dmAPIExec("save,$sessionID,$obj_id");
```

creates and saves a new document in the Docbase with attribute values defined in `%ATTRS`. *Note: dm_CreateObject() does not save the object. You must execute the `save()` method separately.*

```
$obj_id = dm_CreateObject ("dm_document");
$api_stat = dmAPIExec("save,$sessionID,$obj_id");
```

creates and saves a document in the Docbase with no attribute values assigned.

3.5 dm_CreatePath()

dm_CreatePath() allows you to easily create folder hierarchies in a Docbase. *dm_CreatePath()* returns the `r_object_id` of the newly created folder upon success, or `undef` on failure.

3.5.1 Syntax

```
$folder_id = dm_CreatePath(<path>);
```

where `<path>` is a fully qualified path starting at the Docbase root.

3.5.2 Examples

```
$folder_id = dm_CreatePath('/Temp/Db-Documentum/Test');
```

creates the folder `Test` and returns its `r_object_id`. If `/Temp` and `/Temp/Db-Documentum` do not exist, they are also created.

```
@years = ('/Data/Years/1998','/Data/Years/1999',
          '/Data/Years/2000','/Data/Years/2001');
```

```
foreach $year (@years) {
     die dm_LastError($sessionID) unless dm_CreatePath($year);
}
```

creates the folder hierarchy contained in `@years`.

3.6 dm_LocateServer()

The *dm_LocateServer()* subroutine returns a scalar containing the hostname of the server running the Docbase named in its argument. If a server cannot be found for the Docbase, the subroutine returns `undef`.

3.6.1 Syntax

```
$hostname = dm_LocateServer(<docbase name>);
```

where `<docbase name>` is the name of the Docbase.

3.6.2 Examples

```
print "Enter a Docbase name: ";
chomp ($docbase = <STDIN>);
print "$docbase hosted by server " . dm_LocateServer($docbase) . "\n";
```

prints the name of the Docbase entered at the prompt and the name of the server hosting it.

4 Real-World Examples

In this section, I present five real-world examples of programming for Documentum with Perl. The first example is a straightforward script that checks a user's inbox for items. The second is a Perl implementation of Documentum's Interactive DQL Editor (IDQL). (As of version 1.4, this script can be found in the `/etc` directory of the Db::Documentum distribution.) The last three examples are related in that they implement a system for capturing, searching, and viewing documents in the Docbase. The first of these examples is a Docbase configuration script. The second captures news stories from a wire service (e.g., AP), and imports them into the Docbase. The last example is a set of scripts that provide web-based access to the news stories in the Docbase.

4.1 InBox Tickler

This short, but useful script demonstrates how easy programming for Documentum with Perl really is. It is a simple tickler to alert users if they have any queued items in their inboxes.

4.1.1 tickler.pl

```
1    #!/usr/local/bin/perl
2    # tickler.pl
3    # (c) 2000 MS Roth
4
5    use Db::Documentum qw (:all);
6    use Db::Documentum::Tools qw (dm_Connect dm_LastError);
7
8    print "\nDb::Documentum Inbox Tickler\n";
9    print "----------------------------\n";
10
11   # define $DOCBASE, $USER, $PASSWD
12   # logon or die
13   $SESSION_ID = dm_Connect($DOCBASE,$USER,$PASSWD);
14   die "No session ID obtained.\nDocumentum Error was: " .
15   dm_LastError() unless $SESSION_ID;
16
17   # define SQL for counting
18   $DQL = "SELECT COUNT(*) AS cnt FROM dmi_queue_item WHERE name = user";
19
20   # do duery
21   $col_id = dmAPIGet("query,$SESSION_ID,$DQL");
22
23   # if query successful
24   if ($col_id) {
```

```
25        # loop through collection (of 1) and print count
26        while (dmAPIExec("next,$SESSION_ID,$col_id")) {
27            $count = dmAPIGet("get,$SESSION_ID,$col_id,cnt");
28        }
29        dmAPIExec("close,$SESSION_ID,$col_id");
30        print "You have $count items in your inbox.\n";
31    }
32    # if no collection, error
33    else {
34        print "\nNo collection ID obtained.\n";
35        print "Documentum Error was: " . dm_LastError($SESSION_ID);
36    }
37
38    dmAPIExec("disconnect,$SESSION_ID");
39
40    # __EOF__
```

4.1.2 Discussion

The beauty of scripts like `tickler.pl` is that virtually no effort was invested in their production. They can be cobbled together quickly, used once, and discarded with no loss of investment. Or, they can be used and reused with great utility. In addition, because `tickler.pl` is a script (and not a compiled program), it can be changed tomorrow or next week with little effort. Isn't Perl great?

Though this script is pretty simple, it does contain a few points I want to highlight. First, the script does not define `$DOCBASE`, `$USER`, and `$PASSWD`. There are a number of ways to obtain the `$DOCBASE`, `$USER`, and `$PASSWD` if you don't want to hardcode them in the script. You could prompt the user to enter them (as I will demonstrate later), read them from a file, glean them from the OS, or pass them on the command line. I leave this as an exercise for you.

The second point I want to highlight is the loop between lines 26 - 28. This loop accesses the data in the collection returned by the query on line 21. This loop demonstrates the basic construct for accessing data in a collection. You must execute at least one *`next()`* method on a collection to obtain any data from it, and as noted earlier, always *`close()`* your collections. In between, you can *`get()`* any of the attributes you named in your query.

4.1.3 Output

When `tickler.pl` is run, the output looks like this:

```
Db::Documentum Inbox Tickler
----------------------------
You have 13 items in your inbox.
```

Figure 1 - Sample tickler.pl output.

4.2 IDQL

As I'm sure you are aware, the IDQL editor is a very handy Documentum tool. As a test of my Documentum API programming skills, and to exercise the Db::Documentum module, I wrote the

following implementation of the IDQL editor. Since the IDQL editor is only part of the Docbase Administrator in 4i, a local implementation like this one is very handy. `idql.pl` is included in the Db::Documentum distribution archive.

4.2.1 idql.pl

```
1    #! /usr/local/bin/perl -w
2    # idql.pl
3    # (c) 2001 MS Roth
4
5    use Db::Documentum qw(:all);
6    use Db::Documentum::Tools qw (:all);
7    use Term::ReadKey;
8    $VERSION = "1.2";
9
10   logon();
11
12   # ===== main loop =====
13   $cmd_counter = 1;
14   while (1) {
15       print "$cmd_counter> ";
16       chomp($cmd = <STDIN>);
17       if ($cmd =~ /go$/i) {
18           do_DQL($DQL);
19           $DQL = "";
20           $cmd_counter = 0;
21       } elsif ($cmd =~ /quit$/i) {
22           do_Quit();
23       } else {
24           $DQL .= " $cmd";
25       }
26       $cmd_counter++;
27   }
28
29   sub logon {
30       print "\n" x 10;
31       print "(c) 2001 MS Roth. Distributed as part of Db::Documentum\n";
32       print "Db::Documentum Interactive Document Query Language Editor
                 $VERSION\n";
33       print "-----------------------------------------------------------------
                 \n";
34       print "Enter Docbase Name: ";
35       chomp ($DOCBASE = <STDIN>);
36       print "Enter User Name: ";
37       chomp ($USERNAME = <STDIN>);
38       print "Enter Password: ";
39       # turn off display
40       ReadMode 'noecho';
41       chomp ($PASSWD = <STDIN>);
42       # turn display back on
43       ReadMode 'normal';
44
45       # login
46       $SESSION = dm_Connect($DOCBASE,$USERNAME,$PASSWD);
47       die dm_LastError() unless $SESSION;
48
49       my $host = dm_LocateServer($DOCBASE);
50       print "\nLogon to $DOCBASE\@$host successful. Type 'quit' to quit.\n\n";
51   }
52
53   sub do_DQL {
54       my $dql = shift;
55
```

```
56        print "\n\n";
57
58        # do sql and print results
59        $api_stat = dmAPIExec("execquery,$SESSION,F,$dql");
60
61        if ($api_stat) {
62            $col_id = dmAPIGet("getlastcoll,$SESSION");
63
64            # get _count
65            $attr_count = dmAPIGet("get,$SESSION,$col_id,_count");
66
67            if ($attr_count > 0) {
68                # get _names and _lengths
69                @attr_names = ();
70                @attr_lengths = ();
71
72                for ($i=0; $i<$attr_count; $i++) {
73                    push(@attr_names,dmAPIGet("get,$SESSION,$col_id,_names[$i]"));
74                    my $attrlen = dmAPIGet("get,$SESSION,$col_id,_lengths[$i]");
75                    if ($attrlen < 16) { $attrlen = 16; }
76                    push(@attr_lengths,$attrlen);
77                }
78
79                # print attr names
80                for ($i=0; $i<$attr_count; $i++) {
81                    print $attr_names[$i];
82                    print " " x ($attr_lengths[$i] - length($attr_names[$i]) . "
                                  ");
83                }
84                print "\n";
85
86                # print underbars for attr names
87                for ($i=0; $i<$attr_count; $i++) {
88                    print "-" x $attr_lengths[$i] . " ";
89                }
90                print "\n";
91
92                # print attr values
93                $row_counter = 0;
94                while (dmAPIExec("next,$SESSION,$col_id")) {
95                    my $attr_counter = 0;
96                    foreach my $name (@attr_names) {
97                        my $value = dmAPIGet("get,$SESSION,$col_id,$name");
98                        print $value;
99                        print " " x ($attr_lengths[$attr_counter] -
                                      length($value)) . " ";
100                       $attr_counter++;
101                   }
102                   print "\n";
103                   $row_counter++;
104               }
105               print "\n[$row_counter row(s) affected]\n\n";
106               dmAPIExec("close,$SESSION,$col_id");
107           }
108       }
109       print dm_LastError($SESSION,3,'all');
110   }
111
112   sub do_Quit {
113       print "\n\nQuitting!\n\n";
114       dmAPIExec("disconnect,$SESSION");
115       exit;
116   }
117
118   # __EOF__
```

4.2.2 Discussion

The Term::ReadKey module on line 7 is used to hide the password when it is entered later on line 41. If you don't have this module installed, you can easily get it from the CPAN, or just comment-out this line if you prefer not to use it at all; it has no impact on the operation of the script. Note that if you do comment-out line 7, you must also comment-out lines 40 and 43 where the module is actually used.

The main loop in this script is an infinite `while` loop containing three conditional statements. The first condition tests whether the input from `STDIN` contains "go" (Remember, "go" is the signal to IDQL to execute the entered syntax.). If it does, then *`do_DQL()`* is called with the `$DQL` query string. The second condition tests whether the input contains "quit"' If it does, *`do_Quit()`* is called to terminate the script. The third condition is simply the default, and concatenates `STDIN` to the existing `$DQL` variable to build the query statement.

The heart of this script is the *`do_DQL()`* subroutine on lines 53 - 110. Notice that I used the *`execquery()`* method with the `read_query` flag set to `FALSE` to execute the query (line 59). Because I can't anticipate what kind of DQL statement I will receive in `$DQL`, this allows me the most flexibility in processing the statement. For more details regarding *`execquery()`* and the use of the `read_query` flag, see the *Documentum Server Reference Guide*.

Since I don't know what to expect in the `$DQL` variable, I certainly don't know what to expect in the collection the *`execquery()`* returns. Therefore, I must first interrogate the collection object and extract its column names before I can print the results. Line 65 retrieves the number of columns contained in the collection, and line 67 checks that the number of columns is greater than 0. If it's not, then an error occurred. Lines 72 - 77 iterate over the attributes of the collection object and retrieve the names and widths of its columns. Note that these are metadata about the collection--I haven't yet started to process the actual contents of the collection. Lines 79 - 90 print the names of the columns spaced appropriately for their widths. Finally, lines 92 - 106 iterate over the collection, retrieve the query results, print them, and close the collection.

This was a fun and eye-opening exercise. I hope you learned something from it. Interrogating the collection object is insightful and has lots of application. Check the *Documentum Server Reference Manual* for more information regarding a collection objects's other attributes.

4.2.3 Output

Here is output from a sample session using `idql.pl` (edited slightly to conserve space):

```
(c) 2001 MS Roth. Distributed as part of Db::Documentum
Db::Documentum Interactive Document Query Language Editor 1.2
----------------------------------------------------------------
Enter Docbase Name: Docbase-1
Enter User Name: msroth
```

```
Enter Password:

Logon to Docbase-1@Docpage_serv successful. Type 'quit' to quit.

1> select object_name,title from dm_document where owner_name =
'msroth';
2> go


object_name


               title


------------------------------------------------------------------
------------------------------------------------------------------
--------------------------------------------------  -------------
------------------------------------------------------------------
------------------------------------------------------------------
-----------------------------------


getfile.pl

                Intro to DB::Documentum - getfile.pl


show_files.pl

                Intro to Db::Documentum - show_files.pl


dmquery.html

                Intro to Db::Documentum - dmQuery.html


intro-db-dctm.doc

                Intro to Db::Documentum


tickler.pl

                Intro to Db::Documentum - tickler.pl


[5 row(s) affected]


1>_
```

Figure 2 - A sample idql.pl session.

4.3 Config Script

The motivation for developing Db::Documentum was to perform repeatable, customized Documentum installations around the country. These custom installations required creating new

filestores (for distributed content storage), users, ACLS, cabinet/folder hierarchies, methods and procedures, and registering external database tables. Doing this once was tedious; doing it multiple times was out of the question (Remember, this was in the days before the DocApp.). What I needed was a way to automate installations to the point that I could e-mail a script to a system administrator and have him run it.

I found that Perl was the perfect tool for this. It had all the necessary I/O, OS, and logic capabilities to create a robust installation process; it was easy to modify; and it was cross-platform. All it needed was an interface with Documentum. Thus, Db::Documentum was born[7].

The following configuration script isn't nearly as involved as that mentioned above, but it will give you a taste of how easy it is to configure a Docbase using Perl. This script creates a new object type (`news_wire_type`), whose storage is separate from that of the rest of the Docbase; a new user (`listener`); and a cabinet/folder hierarchy. The configurations made by this script are used by the remaining examples in this tutorial.

4.3.1 config.pl

```
1    #!/usr/local/bin/perl
2    # config.pl
3    # (c) 2000 MS Roth
4
5    $| = 1;
6    use Db::Documentum qw(:all);
7    use Db::Documentum::Tools qw(:all);
8    use Term::ReadKey;
9
10   $BASE_CABINET = 'Data';
11   $CONTENT_DIR = "c:\\documentum\\data\\news";
12
13   @FOLDERS = ( "$BASE_CABINET/News_Services",
14                "$BASE_CABINET/News_Services/AP",
15                "$BASE_CABINET/News_Services/AP/1999",
16                "$BASE_CABINET/News_Services/AP/2000",
17                "$BASE_CABINET/News_Services/Reuters",
18                "$BASE_CABINET/News_Services/Reuters/1999",
19                "$BASE_CABINET/News_Services/Reuters/2000");
20
21   %USERS = ( 'listener'  =>  {'client_capability'   =>  '2',
22                               'default_folder'      =>
                                     "/$BASE_CABINET/News_Services",
23                               'description'         =>  'News Listener',
24                               'home_docbase'        =>  'Docbase-1',
25                               'user_group_name'     =>  'admingroup',
26                               'user_os_name'        =>  'listener',
27                               'user_address'        =>  'listener@Docbase-1',
28                               'user_privileges'     =>  '3',
29                               'user_state'          =>  '0'});
30
31   %TYPES = ( 'news_wire_type'  =>  {'news_agency'  =>  'CHAR(32)'} );
32
33   print "\n\n\n\===== CONFIGURE DOCBASE START =====\n\n\n";
```

[7] After completing these installations, I began to investigate releasing my Perl-Documentum integration to the CPAN. I discovered that only weeks before, Brian Spolarich had published the Db::Documentum module. After examining it, I found that we both came to essentially identical solutions--except that his module was UNIX-based and mine was NT-based. I contacted Brain and we collaborated on versions 1.1, 1.2, and 1.3 of the module. Since 1.4, I have been the sole maintainer.

```
34  logon();
35  create_new_object_types();
36  create_new_storage();
37  build_folder_hierarchy();
38  create_users();
39  logoff();
40  print "\n\n\===== CONFIGURE DOCBASE DONE =====\n\n\n";
41  exit;
42
43  sub logon {
44      print "Enter Docbase name: ";
45      chomp($DOCBASE = (<STDIN>));
46      print "Enter dmadmin Username: ";
47      chomp($USERNAME = (<STDIN>));
48      print "Enter Password: ";
49      ReadMode 'noecho';
50      chomp ($PASSWD = (<STDIN>));
51      ReadMode 'normal';
52      $SESSION_ID = dm_Connect($DOCBASE,$USERNAME,$PASSWD);
53      die dm_LastError() unless $SESSION_ID;
54      print "\n\n";;
55  }
56
57  sub logoff {
58      print "\n\nLogging off...\n\n";
59      dmAPIExec("disconnect,$SESSION_ID");
60  }
61
62  sub build_folder_hierarchy {
63      print "Building folder hierarchy...\n";
64
65      foreach (@FOLDERS) {
66          print "\t$_\n";
67          warn dm_LastError($SESSION_ID) unless dm_CreatePath($_);
68      }
69  }
70
71  sub create_users {
72      print "Creating news users...\n";
73
74      foreach my $user (keys %USERS) {
75          print "\t$user\n";
76
77          my %ATTRS = ();
78          my $puser = $USERS{$user};
79
80          # remember these users need OS accounts too!
81          foreach my $attrs (keys %$puser) {
82              $ATTRS{$attrs} = $$puser{$attrs}
83          }
84          $ATTRS{'user_name'} = $user;
85          my $obj_id = dm_CreateObject("dm_user",%ATTRS);
86          warn dm_LastError($SESSION_ID) unless $obj_id;
87          warn dm_LastError($SESSION_ID) unless
88              dmAPIExec("save,$SESSION_ID,$obj_id");
89      }
90  }
91
92  sub create_new_object_types {
93      print "Creating object types...\n";
94      foreach my $type (keys %TYPES) {
95          print "\t$type\n";
96
97          my %attrs = ();
98          my $ptype = $TYPES{$type};
```

```
99
100         foreach my $attr (keys %$ptype) {
101             $attrs{$attr} = $$ptype{$attr}
102         }
103
104         # we can only do this because we know dm_document is the super-type
105         # of all these types.
106         warn dm_LastError($SESSION_ID) unless
          dm_CreateType($type,"dm_document",%attrs);
107     }
108 }
109
110 sub create_new_storage {
111     print "Creating new storage location...\n";
112
113     # Create new location object
114     my %ATTRS = (object_name        => 'news-location',
115                  path_type          => 'directory',
116                  security_type      => 'public',
117                  file_system_path   => $CONTENT_DIR);
118     my $obj_id = dm_CreateObject("dm_location",%ATTRS);
119
120     if ($obj_id) {
121         dmAPIExec("save,$SESSION_ID,$obj_id");
122     } else {
123         warn dm_LastError($SESSION_ID);
124     }
125
126     # Create new filestore object
127     %ATTRS = ();
128     %ATTRS = (name        => 'news-filestore',
129               root        => 'news-location');
130     $obj_id = dm_CreateObject("dm_filestore",%ATTRS);
131
132     if ($obj_id) {
133         dmAPIExec("save,$SESSION_ID,$obj_id");
134     } else {
135         warn dm_LastError($SESSION_ID);
136     }
137
138     # Create new fulltext index object
139     %ATTRS = ();
140     %ATTRS = (index_name        =>  'news-index',
141               store_name        =>  'news-filestore',
142               location_name     =>  'news-location');
143     dm_CreateObject("dm_fulltext_index",%ATTRS);
144
145     if ($obj_id) {
146         dmAPIExec("save,$SESSION_ID,$obj_id");
147     } else {
148         warn dm_LastError($SESSION_ID);
149     }
150
151     # Alter all news_service types to use news_service
152     $sql = "ALTER TYPE news_service_type SET DEFAULT STORAGE
                    'news-filestore'";
153     my $api_stat = dmAPIExec("execquery,$SESSION_ID,'F',$sql");
154
155     if ($api_stat) {
156         $col_id = dmAPIGet("getlastcoll,$SESSION_ID");
157         dmAPIExec("close,$SESSION_ID,$col_id");
158     } else {
159         warn dm_LastError($SESSION_ID);
160     }
161 }
```

```
162
163  # __EOF__
```

4.3.2 Discussion

This script is pretty straightforward in what it does: logs on, creates some new object types, creates a few folders, creates a new user, and logs off (see lines 33 - 41). It's the details that are of interest, so let's examine them.

First, note the use of the data structures in lines 10 -31. These structures make visualizing and maintaining objects in the script much easier. In fact, the ease-of-use and flexibility of data structures in Perl is on of the primary reasons I chose Perl to create my installation scripts. I encourage the use of data structures like these everywhere possible.

The cabinet/folder hierarchy is contained in the list, `@FOLDERS;` the order of the paths is irrelevant. The users are defined in `%USERS`, a hash of hashes (HoH). The HoH makes it clear by inspection what users I am creating and their attribute values. `%TYPES` is also a HoH making it clear the types of objects I will create. In this example I am creating only one new object type (`news_wire_type`) with a single custom attribute (`news_agency`). Notice that this new type is a subtype of `dm_document` and that `dm_document` has been hardcoded into the *dm_CreateType()* function call in the *`create_new_object_types()`* subroutine (line 106). If I were creating a number of new types with different supertypes, I would need to modify the data structure or subroutine to account for them.

The first subroutine encountered after logging on is *`create_new_object_types()`* (lines 92 - 108). This subroutine uses embedded `foreach` loops to iterate over the `%TYPES` HoH and extract each type definition. The outer `foreach` loop iterates over each key (type name) in the `%TYPES` HoH and the inner loop extracts each type's definition into `%attrs`. Line 106 creates the object in the Docbase using `%attrs`. Notice that I used `warn` statements to indicate when errors occur. `die` is probably more appropriate because an unnoticed error in configuration can be disastrous later.

The next step in the configuration is to create a new storage location for the content of our `news_wire_type` objects. This is done by *`create_new_storage()`* on lines 110 - 161. Three things are notable about this subroutine: 1) the objects it creates; 2) the order in which they are created; and 3) the fact that the leaf directory of `$CONTENT_DIR` cannot exist. This subroutine was adapted from Documentum's `primtp.dql` script found in the *DocPage Server System Administrator's Guide*. See the guide for further explanation.

Next, let's examine the creation of the cabinet/folder hierarchy in the Docbase. *`build_folder_hierarchy()`* on lines 62 - 69 creates the cabinet/folder hierarchy by simply iterating over the `@FOLDERS` list and passing each line (containing a path) to *dm_CreatePath()*, where the real work is done.

The final step in the configuration is to create users. This is done by the *`create_users()`* subroutine on lines 71 - 90. This subroutine functions in a manner nearly identical to *`create_new_object_types()`* in that it uses embedded `foreach` loops to iterate over an

HoH to create users. Remember that each Documentum user also requires an OS account.

It's obvious from this example that with a few data structures and a few smart subroutines, you can use Perl to quickly and easily build configuration scripts for Documentum. With some planning, generalization, and more extensive use of data structures, this script can easily be extended to perform maintenance or data migration. Another use for a script of this nature is to setup data in the Docbase for testing and demonstration. I often write a script like this that *seeds* metadata and content in a Docbase at various locations, states, and workflow steps to perform testing or demonstrations.

4.3.3 Output

Here is the output from running `config.pl`. The output is not nearly as important as the changes occurring inside the Docbase.

```
===== CONFIGURE DOCBASE START =====


Enter Docbase name: Docbase-1
Enter dmadmin Username: dmadmin
Enter Password:

Creating object types...
        news_wire_type
Creating new storage location...
Building folder hierarchy...
        Data/News_Services
        Data/News_Services/AP
        Data/News_Services/AP/1999
        Data/News_Services/AP/2000
        Data/News_Services/Reuters
        Data/News_Services/Reuters/1999
        Data/News_Services/Reuters/2000
Creating news users...
        listener


Logging off...


===== CONFIGURE DOCBASE DONE =====
```

Figure 3 - Output from the config.pl script.

4.4 Serial Port Listener

The `listener.pl` script demonstrates how Perl can be used to capture real-time data and store it in Documentum. The real-time data are news stories streamed over an AP serial news feed and into my PC's serial port. The script listens to the serial port, waiting to hear a story delimiter, and then saves the story in the Docbase. This script assumes that you have an AP serial news feed attached to the serial port of your PC (Doesn't everybody?). If you don't, the `serv.pl` script discussed in section 5.4.3 can be used to simulate one. `listener.pl` makes use of the object, user, and folder hierarchy created by the `config.pl` script above.

4.4.1 listener.pl

```
1    #!/usr/local/bin/perl
2    # listener.pl
3    # (c) 2000 MS Roth
4
5    use Win32::SerialPort;
6    use Db::Documentum qw(:all);
7    use Db::Documentum::Tools qw(:all);
8    use Cwd;
9
10   $PORT_NAME = "COM1";
11   $OUTPUT_DIR = cwd() . "\\news";
12   $DM_BASE_CABINET = "/Data/News_Services/AP";
13   $CURRENT_DM_PATH = "";
14   $LAST_DAY = (localtime)[3];
15
16   @MONTHS = qw(January February March April May June July August
17                September October November December);
18
19   $NonASCII   = '\x7f-\xff';      # chars above printable characters
20   $ctrl1      = '\x00-\x09';      # chars below LF
21   $ctrl2      = '\x0b-\x0c';      # chars between LF and CR
22   $ctrl3      = '\x0e-\x1f';      # chars between CR and ASCII
23
24   $DOCBASE = "Docbase-1";
25   $USERNAME = "listener";
26   $PASSWORD = "XXX";
27
28
29   $SESSION_ID = dm_Connect($DOCBASE,$USERNAME,$PASSWORD);
30       die dm_LastError() unless $SESSION_ID;
31
32   # make sure we have a temp dir for saving our files
33   if (! -e "$OUTPUT_DIR") {
34       mkdir "$OUTPUT_DIR","0777";
35   }
36
37   # make sure we have a dm folder for saving files
38   $CURRENT_DM_PATH = create_todays_folder();
39
40   # open Serial Port
41   $Port = new Win32::SerialPort($PORT_NAME) ||
42       die "*** Could not open $PORT_NAME $^E ***\n";
43   $Port->databits(8);
44   $Port->baudrate(1200);
45   $Port->parity("none");
46   $Port->stopbits(1);
47   $Port->handshake("none");
48   $Port->buffers(4096,4096);
49   $Port->write_settings || undef $Port;
50       die "*** Could not write settings to $PORT_NAME ***\n" unless $Port;
51
52   print "\n\nListening to port $PORT_NAME.  Press Control-C to quit.\n";
53
54   # Loop infinitely over open port
55   while (1) {
56       my $gotit = "";
57       $Port->lookclear;
58
59       # use 'AP-NY-99-99-99 9999EST<' as story delimiter
60       $Port->are_match("-re",'AP-NY-\d{1,2}-\d{1,2}-\d{2,4}\s+\d{2,4}\w{3}<');
```

```
61
62      until("" ne $gotit) {
63          $gotit = $Port->streamline;
64          ##### VERY IMPORTANT TO SLEEP #####
65          sleep(1);
66      }
67      my ($match) = ($Port->lastlook)[0];
68
69      # clean up story
70      $gotit =~ s/\n+/\n/sg;          # make it single spaced
71      $gotit =~ s/[$NonASCII]//sg;    # remove non-ASCII chars
72      $gotit =~ s/[$ctrl1]//sg;       # remove low control chars
73      $gotit =~ s/[$ctrl2]//sg;       # remove middle control chars
74      $gotit =~ s/[$ctrl3]//sg;       # remove upper control chars
75
76      # write story to file
77      my $filename = build_filename();
78      print "$PORT_NAME: " . localtime() . " FILE:$filename -> ";
79      open (APFILE,">$OUTPUT_DIR/$filename") ||
80          die "Could not open $OUTPUT_DIR/$filename. $!";
81      print APFILE $gotit;
82      close (APFILE);
83
84      # do Attrs
85      my %Attrs = ();
86
87      # parse title from story
88      # There are several lines that start with ^ and end with < so examine
89      # them all.
90      while ($gotit =~ /\^(.*?)\</g) {
91          my $t = $1;
92          $t =~ s/\'//g;
93          # choose the last one that is not an ed comment
94          if ( ($t !~ /Ed/) && ($t !~ /AP Photo/) ){
95              $Attrs{'title'} = $t;
96              $Attrs{'title'} =~ s/\'//g;  # remove appostrophies
97          }
98      }
99
100     if (! exists $Attrs{'title'}) {
101         $Attrs{'title'} = $filename;
102     }
103
104     $Attrs{'news_agency'} = 'AP';
105     $Attrs{'a_full_text'} = 'TRUE';
106     $Attrs{'object_name'} = $filename;
107
108     # put in docbase
109     # if the day changed on us, create a new folder
110     if ( (localtime)[3] != $LAST_DAY ) {
111         $CURRENT_DM_PATH = create_todays_folder();
112         $LAST_DAY = (localtime)[3];
113     }
114
115     my $obj_id  = dm_CreateObject("news_wire_type",%Attrs);
116     if ($obj_id) {
117         warn dm_LastError($SESSION_ID,3,"all") unless
118             dmAPIExec("setfile,$SESSION_ID,$obj_id,$OUTPUT_DIR/
                        $filename,crtext");
119         warn dm_LastError($SESSION_ID,3,"all") unless
120             dmAPIExec("link,$SESSION_ID,$obj_id,\'$CURRENT_DM_PATH\'");
121         warn dm_LastError($SESSION_ID,3,"all") unless
122             dmAPIExec("save,$SESSION_ID,$obj_id");
123
124         print " OBJECT:$obj_id\n";
```

```
125          unlink "$OUTPUT_DIR/$filename";
126       }
127       else {
128          print "ERROR: " . dm_LastError($SESSION_ID) . "\n";
129       }
130  }
131
132
133  sub build_filename {
134      my ($sec,$min,$hour,$mday,$mon,$year) = (localtime)[0..5];
135      $mon++;
136      $year += 1900;
137      return sprintf("AP-%04d%02d%02d-%02d%02d%02d\.txt",
                        $year,$mon,$mday,$hour,$min,$sec);
138  }
139
140  sub logoff {
141      print "\n\nLogging off...\n\n";
142      dmAPIExec("disconnect,$SESSION_ID");
143      if (defined $Port) {
144          $Port->close || die "Could not close $PORT_NAME.\n";
145          undef $Port;
146      }
147      exit;
148  }
149
150  sub create_todays_folder {
151      # create today's folder in docbase
152      my ($mday,$mon,$year) = (localtime)[3..5];
153      $mday = "0$mday" if ($mday !~ /\d\d/);
154      $year += 1900;
155      my $path = "$DM_BASE_CABINET/$year/$MONTHS[$mon]/$mday";
156      die dm_LastError($SESSION_ID) unless dm_CreatePath($path);
157      return $path;
158  }
159
160  # __EOF__
```

4.4.2 Discussion

`listener.pl` uses the Win32::SerialPort module. Win32::SerialPort provides an object-based interface to your PC's serial port[8]. If you don't have this module, you need to get it from the CPAN. For more information regarding the use of Win32::SerialPort, see the Win32::SerialPort documentation.

To begin, this script defines some constants (lines 10 - 26), logs on to the Docbase (lines 29 - 30), creates a local working directory (lines 32 - 35), creates a folder hierarchy in the Docbase (lines 37 - 38), and opens and configures the serial port (lines 40 - 52). Then, this script lives within the infinite `while` loop defined on lines 54 - 130. Each story is read from the serial port by the `until` loop on lines 62 - 66. When the story delimiter (defined on line 60) is encountered in the input data, the story is assigned to `$gotit` (line 63) and the `until` loop exits. *Note: the `sleep` in this loop is paramount. Without it, the script won't give any CPU time to other processes on your computer!*

Once a complete story is contained in `$gotit`, it is cleaned up on lines 69 - 74 and saved to a

[8] The UNIX equivalent of Win32::SerialPort is Device::SerialPort; also available from the CPAN.

temporary file on lines 76 - 82. Lines 87 - 106 extract information from the body of the story itself and assign values to a few attributes in the `%Attrs` hash. Lines 108 - 129 check the story into the Docbase using *`setfile()`* to transfer the story contents and *`link()`* to put it in the correct folder. That's essentially it.

There are a few things I want to point out about this script. First, examine lines 87 - 98. This loop tries to extract the title from the story. Legitimate news stories have a title, and often an editor and photographer that are denoted by lines that begin with "^" and end with "<". For example:

```
^Suspect in police shooting agrees to testify for lighter sentence<
^AP Photos<
```

When that is the case, and the reporter followed the format, this loop does a good job of extracting the title and assigning it to the `title` key in `%Attrs`. However, the news wire often carries stories that don't adhere to this format such as: corrections, multi-part stories, announcements, and stock market reports. In those cases, no title is extracted and `$filename` is assigned to the `title` key of `%Attrs`.

Second, notice that on line 105 I explicitly flag the story for full-text indexing. This is important if you want to be able to find the story using the web interface in the next example. This, of course, assumes that your server is running the full-text indexing job.

This is a powerful script for only 160 lines of code. It contains the basic elements needed for real-time and bulk data loading, and it wouldn't take much to convert it from a script that listens to a serial port to one that reads a file or a database table as input. Scripts of this type are invaluable when bulk loading or migrating data into Documentum.

I have found that `listener.pl` runs well as a Windows NT service. To learn how to convert a Perl script into a Windows NT service, read Kevin Meltzer's *Perl Journal* (*http://www.tpj.com*) article "Turning a Perl Program Into an NT Service" in issue #15.

4.4.3 serv.pl

`listener.pl` makes a big assumption. It assumes that you have access to an AP serial news feed. I realize that most people do not have access to such a thing. In fact, I didn't either when I wrote this script. That's why I wrote the `serv.pl` script. `serv.pl` will serve any text file to a serial port. You can think of it as the reciprocal of `listener.pl` in that it *talks* instead of *listens* to the serial port!

```
1    #!/usr/bin/perl
2    # serv.pl
3    # (c) 2000 MS Roth
4
5    use Win32::SerialPort;
6
7    $PORT_NAME = "COM2";
8    $DATA_FILE = "ap.txt";
```

```
9
10   # open port
11   $Port = new Win32::SerialPort($PORT_NAME) ||
                 die "Could not open $PORT_NAME. $^E\n";
12   $Port->databits(8);
13   $Port->baudrate(1200);
14   $Port->parity("none");
15   $Port->stopbits(1);
16   $Port->handshake("none");
17   $Port->buffers(4096,4096);
18   $Port->write_settings || undef $Port;
19       die "Could not write settings to $PORT_NAME.\n" unless $Port;
20
21   # read data file into memory
22   open(DATA,"<$DATA_FILE") || die "Could not open $DATA_FILE for read. $!";
23   @data = (<DATA>);
24   close(DATA);
25
26   print "\n\nServing test data ($DATA_FILE) on port $PORT_NAME.
            Press Control-C to quit.\n\n";
27
28   # serve data
29   while(1) {
30       print "\n#";
31       foreach (@data) {
32           $Port->write($_);
33           if (/AP-NY/i) {
34              print "\.";
35              sleep(int(rand 20));
36           }
37       }
38   }
39
40   # __EOF__
```

4.4.4 Discussion

This script simply opens the serial port, reads the `$DATA_FILE` into memory, and endlessly spews it out. A few things to note here:

1. My `ap.txt` was a text file containing real, captured data from my customer's AP serial news feed. You can use any text file that you like, but you will want to change the regular expression on line 33 to match your story delimiter. You will need to make this change in `listener.pl` also (line 60).
2. Line 35 executes a random `sleep` between successive stories. This pause between stories more closely emulates the timing of a real AP serial news feed.
3. To use `listener.pl` and `serv.pl` together on the same machine, run them in separate command windows, and connect your serial ports together with a null modem cable.[9]

4.4.5 Output

The output generated by the `serv.pl` script looks like this. Each "`#`" represents an iteration of the data file, and each "`.`" represents a story sent to the serial port.

[9] This should also work on UNIX systems, although I confess, I have never tried it.

```
Serving test data (ap.txt) on port COM2.  Press Control-C to quit.
#. . . . . .
. . . . . .
#. . . . . .
```

Figure 4 - Output from serv.pl script.

The output generated by the `listener.pl` script is a little more interesting. `listener.pl` informs you at which port and at what time a story was received, what it named the file (and hence the `object_name` in the Docbase), and what the `r_object_id` was when it was checked into the Docbase.

```
Listening to port COM1.  Press Control-C to quit.
COM1: Wed Jul  5 10:51:51 2000 FILE:AP-20000705-105151.txt ->
OBJECT:0901605380011120
COM1: Wed Jul  5 10:52:19 2000 FILE:AP-20000705-105219.txt ->
OBJECT:0901605380011121
COM1: Wed Jul  5 10:52:43 2000 FILE:AP-20000705-105243.txt ->
OBJECT:0901605380011122
COM1: Wed Jul  5 10:52:58 2000 FILE:AP-20000705-105258.txt ->
OBJECT:0901605380011123
COM1: Wed Jul  5 10:53:16 2000 FILE:AP-20000705-105316.txt ->
OBJECT:0901605380011124
COM1: Wed Jul  5 10:53:25 2000 FILE:AP-20000705-105325.txt ->
OBJECT:0901605380011125
COM1: Wed Jul  5 10:53:33 2000 FILE:AP-20000705-105333.txt ->
OBJECT:0901605380011126
```

Figure 5 - Sample listener.pl output.

In WorkSpace, the result of running `config.pl` and `listener.pl` looks like this:

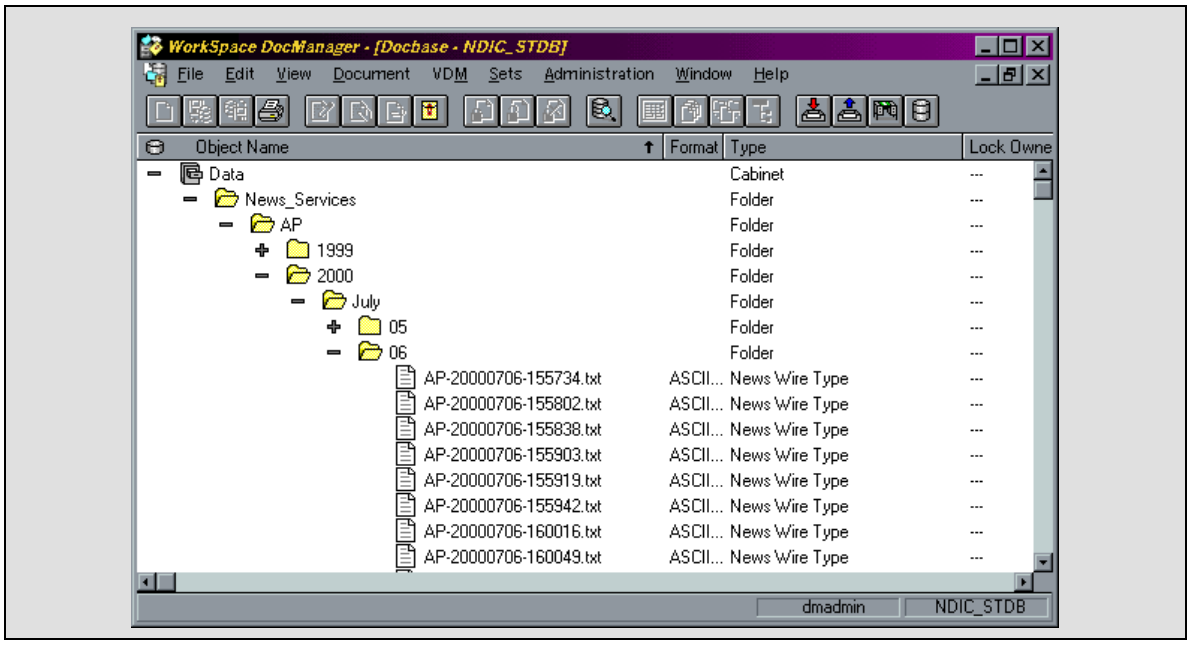

Figure 6 - Cabinet/folder hierarchy in WorkSpace.

4.5 Web Access

One of the more fun applications of Db::Documentum is the World Wide Web. Documentum provides a web server (RightSite[10]) and environment for developing and fielding web-based solutions. However, sometimes all that overhead isn't necessary. Sometimes, a simple solution will suffice. In these instances, Perl and Db::Documentum fit the bill perfectly.

The following set of scripts provide a bare-bones example of hosting a Docbase on the web using CGI, Perl, and Db::Documentum. The scripts will provide web-based search-and-retrieval services for the archive of news stories the `listener.pl` script is depositing in the Docbase. The first script, `show_files.pl`, logs on; querys the Docbase for news stories; and builds a list of the query results. The second script, `getfile.pl`, displays the files on the query results list when they are selected.

This example assumes you are running a web server. If not, there are plenty of free ones on the web--some are even written in Perl (check the CPAN). Both `show_files.pl` and `getfile.pl` run as CGI programs and should be located in your web server's `cgi-bin` directory.

4.5.1 dmquery.html

Before these scripts will run, they need an HTML page to launch from; `dmquery.html` provides that launch pad. This page displays a simple form that asks you to enter your logon

[10] And now with Documentum 4i v4.1, the Web Developers Kit (WDK).

information and a word to search for.

```
1    <HTML>
2    <!-- dmquery.html -->
3    <!-- (C) 2000 MS Roth -->
4    <HEAD><TITLE>Db::Documentum Web Demo Page</TITLE></HEAD>
5
6    <BODY>
7    <P>
8    <CENTER>
9    <H1>Db::Documentum Web Demo Page</H1>
10   <H2>Logon to Docbase</H2><P>
11   <FORM METHOD=POST ACTION='cgi-bin/show_files.pl'>
12   <TABLE BORDER=1>
13   <TR><TD>Docbase</TD><TD><INPUT TYPE='text' SIZE=40 NAME='docbase'></TD></TR>
14   <TR><TD>User</TD><TD><INPUT TYPE='text' SIZE=40 NAME='username'></TD></TR>
15   <TR><TD>Password</TD><TD><INPUT TYPE='password' SIZE=40
     NAME='password'></TD></TR>
16   </TABLE>
17   <H2>Query Docbase for:</H2>
18   <INPUT TYPE='text' NAME='search'>
19   <INPUT TYPE='submit'>
20   </FORM>
21   </CENTER>
22   </BODY>
23   </HTML>
```

4.5.2 show_files.pl

The `show_files.pl` script is the `action` of the form created by `dmquery.html`. It does the query and displays the results as hyperlinks.

```
1    #!/usr/bin/perl
2    # show_files.pl
3    # (c) 2000 MS Roth
4
5    $| = 1;
6    use Db::Documentum qw(:all);
7    use Db::Documentum::Tools qw(:all);
8    use CGI qw(:standard);
9    use CGI::Carp 'fatalsToBrowser';
10
11   $q = new CGI;
12
13   # get input from form
14   $input    = $q->param('search');
15   $docbase  = $q->param('docbase');
16   $username = $q->param('username');
17   $password = $q->param('password');
18
19   # logon to Docbase
20   $session = dm_Connect($docbase, $username, $password);
21   die dm_LastError() unless $session;
22
23   # set cookie for use by getfile.pl
24   %dm_session = ('docbase' =>$docbase,
25                  'username'=>$username,
26                  'password'=>$password);
27
28   $cookie = $q->cookie(-name    => 'dm_session',
29                        -value   =>\%dm_session,
30                        -path    => '/cgi-bin',
```

```
31                        -expires => '+5m');
32
33   print $q->header(-cookie => $cookie);
34
35   # start html page
36   print $q->start_html();
37   print "<CENTER><H2>Search Docbase for title containing \'$input\'
       </H2> </CENTER><HR>\n";
38
39   # build DQL query string
40   my $DQL = "select title,r_object_id,r_creation_date,news_agency from
                news_wire_type search document contains \'$input\' order
                by r_creation_date";
41
42   # do query
43   $col_id = dmAPIGet("readquery,$session,$DQL");
44
45   # if query succeeded
46   if ($col_id) {
47       print "<UL>\n";
48       # iterate over collection and print links
49       while(dmAPIExec("next,$session,$col_id")) {
50           my $title = dmAPIGet("get,$session,$col_id,title");
51           my $obj_id = dmAPIGet("get,$session,$col_id,r_object_id");
52           my $date = dmAPIGet("get,$session,$col_id,r_creation_date");
53           my $agency = dmAPIGet("get,$session,$col_id,news_agency");
54           print "<LI><A HREF=\"cgi-bin/getfile.pl?obj_id=$obj_id\">$date --
                     $agency -- $title</A></LI>\n";
55       }
56       dmAPIExec("close,$session,$col_id");
57       print "</UL>\n";
58   } else {
59       print "<CENTER><H3>Query returned no results.</CENTER>\n";
60   }
61
62   print $q->end_html();
63
64 # __EOF__
```

4.5.3 Discussion

There are three aspects of this script I want to discuss. The first is the use of CGI.pm on lines 8 and 9: don't do CGI without it. The second is the form of the URL printed on line 54. This URL points to the `getfile.pl` script and passes (via the query-info portion of the URL) the object ID of the document to retrieve. For example, an `<A>` tag produced by line 54 might look like this:

```
<A HREF="cgi-bin/getfile.pl?obj_id= 0901653800111d8">.
```

Third, I want to discuss the cookie. Lines 23 - 33 save your logon information to a cookie. This is necessary because Db::Documentum executes a *dmAPIDeInit()* automatically when `show_files.pl` terminates. *dmAPIDeInit()* destroys the `apiconfig` object created by *dmAPIInit()* and effectively closes the session (RightSite certainly has an advantage over us here!). To retrieve a document from the Docbase, `getfile.pl` will need an active session. To provide it one, I stash your logon information in the cookie and retrieve it in `getfiles.pl` to logon again.

Now, I will be the first to admit that saving logon information--especially passwords--in a cookie

is a *bad* idea. There are better solutions. However, since this script is meant only for demonstration, I took the simple approach and saved the logon information in the cookie. PLEASE DON'T IMPLEMENT A SYSTEM LIKE THIS, IT'S DANGEROUS!

4.5.4 getfile.pl

The `getfile.pl` script retrieves the content of the document selected from the search results page and displays it.

```
1    #!/usr/bin/perl
2    # getfile.pl
3    # (c) 2000 MS Roth
4
5    $| = 1;
6    use Db::Documentum qw(:all);
7    use Db::Documentum::Tools qw(:all);
8    use CGI qw(:standard);
9    use CGI::Carp 'fatalsToBrowser';
10
11   $q = new CGI;
12
13   # get obj_id from command line
14   $obj_id = $q->param('obj_id');
15
16   # get session from cookie
17   %dm_session = $q->cookie(-name => 'dm_session');
18   $docbase  = $dm_session{'docbase'};
19   $username = $dm_session{'username'};
20   $password = $dm_session{'password'};
21
22   # logon or die
23   $session = dm_Connect($docbase, $username, $password);
24   die "No session or session has expired\." unless $session;
25
26   # get the type of content associated with obj_id
27   $DQL = "select a_content_type,object_name from dm_document where
         r_object_id = \'$obj_id\'";
28   $col_id = dmAPIGet("readquery,$session,$DQL");
29   if ($col_id) {
30       while(dmAPIExec("next,$session,$col_id")) {
31           $type = dmAPIGet("get,$session,$col_id,a_content_type");
32           $title = dmAPIGet("get,$session,$col_id,object_name");
33       }
34       dmAPIExec("close,$session,$col_id");
35   }
36   else {
37       $type = 'crtext';
38   }
39
40   # if plain text, print
41   if ($type eq 'crtext') {
42       print $q->header();
43       print $q->start_html();
44       print "<CENTER><H2>$title</H2></CENTER><HR>\n";
45
46       $col_id = dmAPIGet("getcontent,$session,$obj_id");
47       while(dmAPIExec("next,$session,$col_id")) {
48           $doc .= dmAPIGet("get,$session,$col_id,_content_buffer");
49       }
50       dmAPIExec("close,$session,$col_id");
51
52       print "<PRE>$doc</PRE>\n";
53       print $q->end_html();
```

```
54   }
55   # if other than plain text, download it to web server and redirect
56   else {
57       $dl_file = dmAPIGet("getfile,$session,$obj_id,\.\.\\$obj_id\.$type");
58       print $q->redirect("http://<host>/temp/$obj_id\.$type");
59   }
60
61   # __EOF__
```

4.5.5 Discussion

The `getfile.pl` script is simple enough. The basic idea is to get the object ID from the URL, query the Docbase to determine what type the object is (text vs. binary), and retrieve the content of the object in an appropriate manner.

The script begins by retrieving the object ID from the URL (line 14) and the cookie from the browser (lines 17 - 20). Both of these tasks are made possible by CGI.pm. Lines 22 - 38 logon to the Docbase and execute a query to retrieve the content type information about the document referenced by `$object_id`.

Starting at line 40, I branch based upon the content of `$type`. If I am dealing with a text document (`crtext`), I execute lines 42 - 54, which use *`getcontent()`* to retrieve the object's content and print it. Notice the loop on lines 47 - 50. This is necessary because *`getcontent()`* returns a collection of pages. For this example, the collection will contain only one page.

If the object's type is not text, the `else` condition is executed on lines 55 - 59. This is a bit over designed for this example because I am only ever retrieving text files (`news_wire_type`), but I wanted to show you an interesting trick if you ever expect to retrieve binary files. Line 57 downloads the file from the Docbase to the web server. Line 58 then redirects the browser to `$dl_file` and lets the server and the browser resolve the MIME type and launch the appropriate plug-in or helper application. Pretty neat trick, eh?

Two notes:

1. You will need to modify the path and host names on lines 57 - 58 to reflect your configuration.
2. If you do retrieve binary files, make sure you occasionally delete the `$dl_file` files from your web server; they can really add up after a while.

4.5.6 Output

To use the web interface, point your browser at *http://<host>/dmquery.html*, enter your logon information and a word to search on.

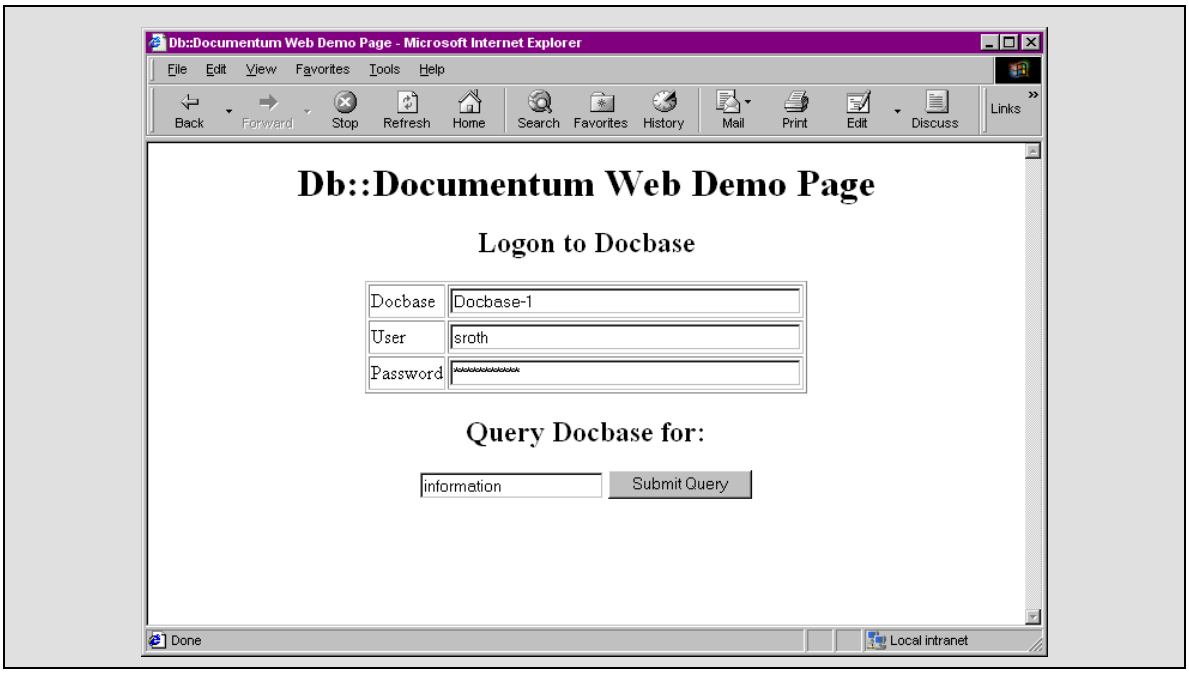

Figure 7 - dmquery.html.

Clicking the Submit Query button results in the following response from `show_files.pl`

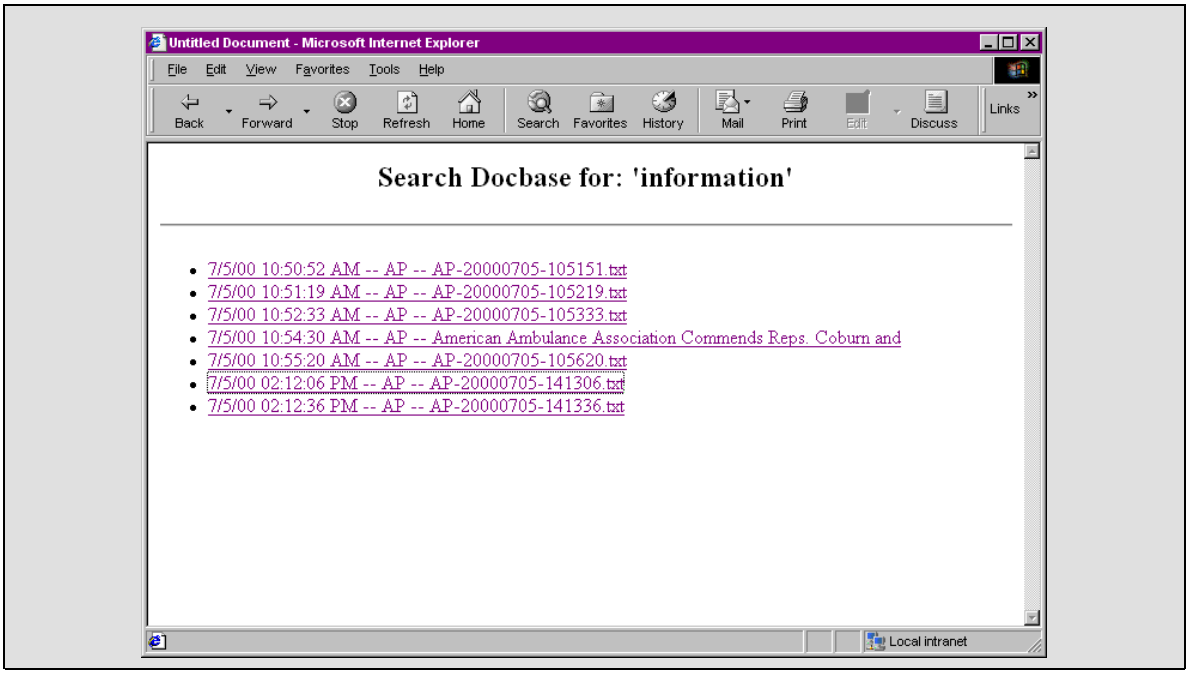

Figure 8 - query results for 'information.'

Clicking on "American Ambulance Association Commends Reps. Coburn and" launches `getfile.pl` to display its contents.

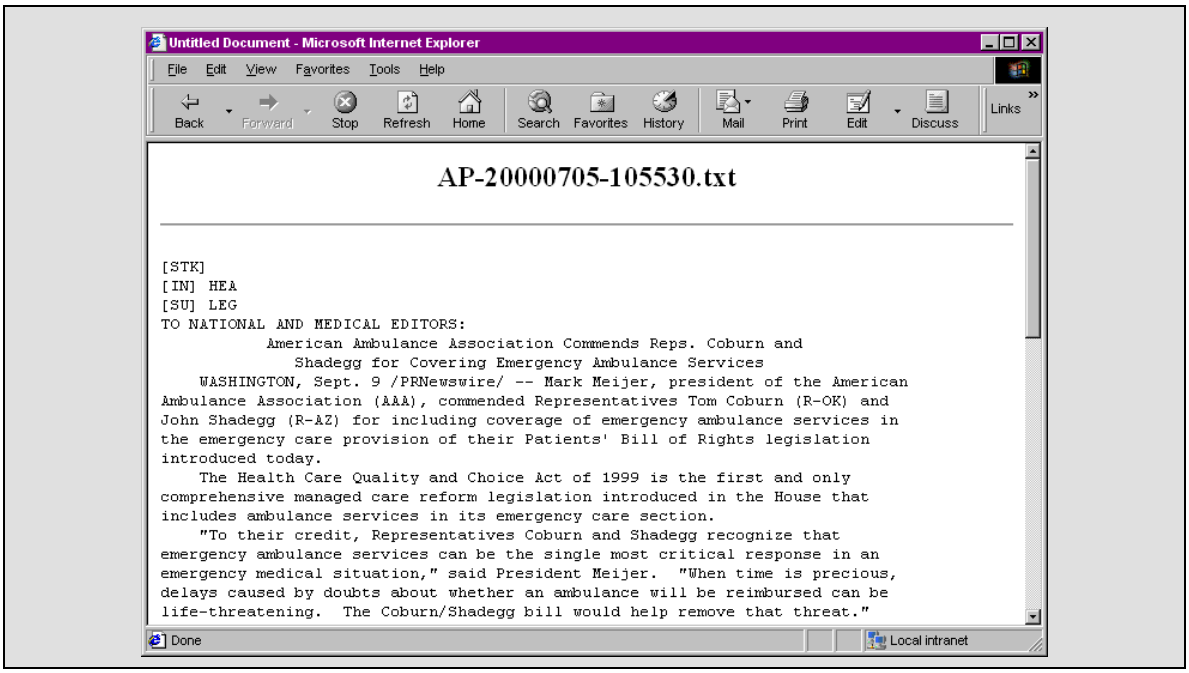

Figure 9 - getfile.pl.

This is a very simple example to demonstrate the basics of using CGI, Perl, and Db::Documentum to host a Docbase on the web, it is far from complete. Besides the security issue with the cookie, you might consider:

- There is little or no error checking or recovery in either Perl script. If empty collections or object IDs are returned, you are often presented with only a blank page in your browser.
- In the `show_files.pl` script, you could display the cabinet/folder path associated with each `$title` by querying for the `i_folder_id` and associating it with the corresponding `dm_folder` object. This would provide the user some additional context (i.e., date and wire service name) about the story.
- You could try using tickets instead of passwords in the cookie.
- You could give everyone anonymous access to the Docbase, do away with the cookie, and hard code the logon in the script.
- You could improve the DQL query to process Boolean search terms.
- You could employ templates to display the news articles in a more pleasing style and add navigational aids so the directory structure can be traversed.
- You could use the ideas here to create an entirely different web application. For instance, a guest book. The possibilities abound!

5 Closing

This tutorial has discussed the installation and contents of the Db::Documentum module and how to use it. Through the use of the real-world examples, I hope I have given you an appreciation of how easy and powerful programming for Documentum with Perl can be. From simple, one-time scripts, to data capturing and migration, to entire applications, Perl and Db::Documentum can do it all. Not only that, they can do it easily, flexibly, and cross-platform.

I know that Perl and Db::Documentum lack the flash and glamour of other Documentum-enabled programming languages (e.g., Visual Basic, Java), but they excel in power and simplicity. Besides, real hackers like 80x24! Who needs a GUI?

I would like to express my thanks to the people who have assisted me with this tutorial: Frances, Matt, John, and Scott. Thanks!

[*SOLI DEO GLORIA*]

M. Scott Roth is co-author of the Db::Documentum module, and "just another Perl hacker" at SAIC in Reston, VA. Mr. Roth has also built a Perl interface to the DFC (Db::DFC) available from the CPAN. Feel free to contact him with your comments, suggestions, complaints, etc. regarding this tutorial or Db::Documentum in general. Mr. Roth can be reached at michael.s.roth@saic.com.