EMu Documentation

FIFO Server

Document Version 1

EMu Version 4.0

Contents

SECTION 1	**FIFO Server**	**1**
	Overview	1
SECTION 2	**Invoking the FIFO server**	**3**
	Scripts and command line	4
	KE Texpress Validation	5
	C++ Client Code	6
SECTION 3	**FIFO Server plugins**	**7**
SECTION 4	**Standard plugins**	**13**
	Centroid	13
	System Lookup	13
	Index	**15**

SECTION 1

FIFO Server

Overview

One issue with using a generic database server is that values often need to be computed or processes invoked when saving a record. In many cases these computations occur within the confines of the database server itself. In the case of KE Texpress (the database engine used by EMu) a powerful scripting language allows values to be computed and data adjusted when a record is saved. When a command needs to be run, the `system()` call may be used. In the case where complex data computations are required, particularly where the computed values are the result of other table lookups, the `system()` call provides a useful solution. The format of the call is:

```
result = system("command");
```

The `"command"` provided as the argument is run and the output returned. Where values are computed, the output is assigned to one or more columns. If a process is invoked (for example printing a specimen label), the output may be empty. The `system()` call provides a useful mechanism for allowing external programs to be invoked.

Similar functionality is available through the C++ TexVCL objects used by the client. The **KESession** object provides the `Execute()` method:

```
status = Session->Execute("command", output, error);
```

where:

`command`	is the program to be executed.
`status`	is the exit status of the command (zero indicates the command completed successfully).
`output`	is an **AnsiString** that receives any output sent to *stdout*.
`error`	is an **AnsiString** that receives any output sent to *stderr*.

Most script languages provide a mechanism for invoking commands from within the script itself and capturing the output (e.g. perl has a `system()` call and also provides back ticks).

The problem with invoking a command to start a process or to generate values is the expense in terms of computing resources. Each command invoked needs to learn about its environment (e.g. if a database table is consulted, the table schema needs to be loaded each time the command is called). Also, due to the way commands are started in a UNIX environment (via the `fork()` call), commands invoked by large programs (e.g. texserver) start up slowly. It is this combination of slow start up and the constant reloading of the environment that may result in a high load being placed on the server machine.

The FIFO service was introduced in KE EMu 4.0.01 to address these two issues:

- The first is addressed by removing the need to start a new command. Instead the FIFO server provides mechanisms where KE Texpress, C++ client code and scripts can ask the FIFO server to perform some function. Similar to command execution, the return value is sent back to the caller. The big difference however is that the FIFO server is running all the time, rather than starting each time a request is made. This removes the need for a command to be started.
- The second issue is addressed by the FIFO server providing access to resident database servers, rather than starting a new database server each time data is required. As the database servers are resident, the schema is only read when the database server is first loaded.

Using the FIFO server dramatically increases the rate at which commands can be executed, while lowering the overall load placed on the server. The FIFO server has been designed to be extensible by adding plugins that provide new functionality.

SECTION 2

Invoking the FIFO server

The FIFO server requires two pieces of information and returns one. The two pieces of information required as input are:

plugin The name of the function inside the FIFO server to be invoked. The name can consist of any characters except a newline character. It is normally descriptive (e.g. *Centroid* to invoke the centroid calculator).

data Information forwarded to the plugin used to compute values or start processes. For example the data:

```
DMS|14 45 12 N|43 15 W
DEC|14.7845|-43.27
```

supplied to the *Centroid* plugin provides two latitude / longitude points (the first set in Degrees / Minutes / Seconds and the second as a decimal number of degrees) for which the centroid is to be calculated. The format of the input *data* is plugin dependent.

The information returned is the computed value. The format of the data returned depends on the plugin invoked. For example, the value returned for the above *Centroid* input is:

```
14 46 8.1 N|43 15 36 W|14.76892|-43.260
```

where the first two values are the centroid expressed as DMS, and the next two expressed as decimal degrees.

Only one instance of the FIFO server runs for a given client environment. All users access this instance when requests are made of the FIFO server. In order to ensure that all requests are handled serially, a simple file locking mechanism is used. This guarantees that the correct output is received for the input provided.

The FIFO server is installed as a background load. The `emuload` command is used on the server to control access:

To start the FIFO server use:

```
emuload start fifo
```

To check the status of the server use:

```
emuload status fifo
```

To stop the server use:

```
emuload stop fifo
```

Scripts and command line

The command `emufifo` is used to invoke the FIFO server from the command line or from within a script. Its usage message is:

```
Usage: emufifo plugin [data]
where
   plugin           name of fifo plugin to invoke
   data             data passed to fifo plugin [default: stdin]
```

For the *Centroid* plugin example above, the following command could have been used:

```
emufifo Centroid << EOF
DMS|14 45 12 N|43 15 W
DEC|14.7845|-43.27
EOF
14 46 8.1 N|43 15 36 W|14.76892|-43.260
```

The data may also be supplied as an argument:

```
emufifo Centroid "DMS|14 45 12 N|43 15 W
DEC|14.7845|-43.27"
14 46 8.1 N|43 15 36 W|14.76892|-43.260
```

giving the same response. `emufifo` is often used to debug new plugins.

KE Texpress Validation

The FIFO server may also be invoked from within KE Texpress. When a record is saved, a validation handler is run. The handler checks for consistent data but may also be used to compute values. The following code segment shows how to invoke the FIFO server from within the validation handler:

```
/*  FIFO settings.
*/
fifoin = getenv("EMUPATH") . "/loads/fifo/input";
fifoout = getenv("EMUPATH") . "/loads/fifo/output";
fifolock = getenv("EMUPATH") . "/loads/fifo/lock";

/*  "System Yes" value
*/
if ((YES = getenv("SYSYES")) == "")
{
        YES = fifo(fifoin, fifoout, fifolock, "System
Lookup\nSystem Yes");
        setenv("SYSYES", YES);
}
```

The `fifo()` call is used to communicate with the FIFO Server. Its arguments are:

`fifoin`	The path to the input side of the FIFO server. The name of the plugin and data are written to this file (the file is actually a named pipe created when the server is invoked).
`fifoout`	The path to the output side of the FIFO server. The results are read from this file (the file is also a named pipe created when the server is invoked).
`fifolock`	The path to an empty file used as a lock to ensure that only one process can access the FIFO server at a time. The locking ensures that correct results are returned for a given request.
`fifovalue`	The information to be forwarded to the FIFO server. The first line must contain the name of the plugin to be invoked. All remaining lines are passed to the plugin as data.

The code above calls the *System Lookup* plugin (which returns the value associated with the name of the lookup list supplied as data), asking for the value of the *System Yes* table. The returned value is remembered so it only needs to be looked up once. The values for `fifoin`, `fifoout` and `fifolock` defined above should always be used. Care should be taken with values returned by the FIFO server. In many cases the return value may have a trailing newline character that may need to be removed (in the case of the *System Lookup* plugin this is not the case, but it is for the *Centroid* plugin).

C++ Client Code

A new method has been added to the **KESession** object that communicates with the FIFO server. The method is:

```
AnsiString
__fastcall
KESession::Fifo(AnsiString fifoin, AnsiString fifoout, AnsiString
fifolock, AnsiString fifovalue)
```

The arguments are the same as for the Texpress validation call. The return value is the information sent back from the FIFO server. In order to provide easier access to the server, a new method has been added to the base window class **TBaseFrm** that invokes `Fifo()` with the correct paths. The method is:

```
AnsiString
__fastcall
TBaseFrm::FifoServer(AnsiString plugin, AnsiString data)
```

The simplified version only requires the name of the plugin to invoke and any data to be passed to it. For example to get the value of the *System Yes* lookup list, the following call could be used:

```
AnsiString results = FifoServer("System Lookup", "System Yes");
```

SECTION 3

FIFO Server plugins

The FIFO server is designed to be extensible. In fact the server itself just provides a framework without any services built in. All services are provided by **plugins** that are loaded when the FIFO server is started. A plugin is really just a perl library that contains a registration function used to define what plugin types are handled. The standard plugins are located in **etc/fifo**, while client specific plugins can be found in **local/etc/fifo**. When the FIFO server starts it looks in both the standard and local directories for all files with a **.pl** extension (a perl library). The file is loaded and the `Register()` function invoked to determine what plugins are handled by the script.

The shell of a plugin looks like:

```
#!/usr/bin/perl
#
#  Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
no warnings 'redefine';

#
#  Registration function.
#
sub
Register
{
        my $plugins = shift;

        #
        #  We handle the "Plugin Name" method.
        #
        $plugins->{"Plugin Name"} = \&Plugin;
}

#
#  The handler for the "Plugin Name" plugin
#
sub
Plugin
{
        my $plugin = shift;
        my $data = shift;
```

```
        #
        #  Plugin code
        #
}

1;
```

The `Register` subroutine is called by the FIFO server passing in a reference to a hash. It is necessary to add the following to the hash:

- The name of the plugin to be handled.
- A reference to the function to invoke when the plugin is called.

As many different handlers as required may be registered within the one plugin. When a call is made to the FIFO server and the plugin name matches the one registered, the corresponding plugin subroutine is called. Two arguments are supplied to the plugin subroutine:

- The plugin name that matched.
- A reference to a list of input lines (where the newline has been removed from each line).

Any value returned by the handler is sent back to the caller.

An example may make things clearer. Let's create a local plugin that provides two functions:

- Addition, which will add up all the numbers supplied as data.
- Multiplication, which will multiply the numbers supplied.

The plugin will be placed in **local/etc/fifo/math.pl**. The code is:

```
#!/usr/bin/perl

#
#  Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
no warnings 'redefine';

#
#  Registration function.
#
sub
Register
{
        my $plugins = shift;

        #
        #  We handle the "Addition" and “Subtraction” methods.
```

```
        #
        $plugins->{"Addition"} = \&Addition;
        $plugins->{"Multiplication"} = \&Multiplication;
}

#
#  The handler for the "Addition" plugin
#
sub
Addition
{
        my $plugin = shift;
        my $data = shift;
        my $total = 0;

        #
        #  Add up the values supplied
        #
        foreach my $value (@{$data})
        {
                $total += $value;
        }
        return($total);
}

#
#  The handler for the "Multiplication" plugin
#
sub
Multiplication
{
        my $plugin = shift;
        my $data = shift;
        my $total = 1;

        #
        #  Add up the values supplied
        #
        foreach my $value (@{$data})
        {
                $total *= $value;
        }
           return($total);
}

1;
```

Notice that the `Register` subroutine adds two handlers (one for `Addition` and one for `Multiplication`). Associated with each handler is the subroutine to call when

the handler is matched (`Addition` and `Multiplication` respectively). Next we restart the FIFO server (using `emuload stop fifo` and `emuload start fifo`). Now we can use `emufifo` to test our plugin:

```
emufifo Addition << EOF
1
2
3
4
EOF
10
emufifo Multiplication << EOF
1
2
3
4
EOF
24
```

Although the example above is trivial it does present the basics involved in setting up a new plugin. In many cases the plugin needs to access data stored in a KE Texpress table. In this case the plugin can use either `OldServer()` to get a reference to a **texql** object (as defined in `texql.pm`) or `NewServer()` for a reference to a **texapi** object (as defined in `texapi.pm`). Using either of these objects you can retrieve data from existing tables and use it to build the result. As an example the plugin below determines whether a given IRN has any child records:

```
#!/usr/bin/perl

#
#  Copyright (c) 1998-2009 KE Software Pty Ltd
#

use strict;
use warnings;
no warnings 'redefine';

#
#  Registration function.
#
sub
Register
{
        my $plugins = shift;

        #
        #  We handle the "Has Children" method.
        #
        $plugins->{"Has Children"} = \&Children;
```

```
}

#
#  The handler for the "Has Children" plugin
#
sub
Children
{
        my $plugin = shift;
        my $data = shift;
        my $texql;
        my $row;

        #
        #  Check if we have any records.
        #
        $texql = OldServer();
        $texql->Command(
            "select all from ecatalogue where ParParentRef = " .
$data->[0]);

        #
        #  Get the result
        #
        $row = $texql->Row();
        $texql->Finish();

        #
        #  Return the result
        #
        return(defined($row) ? "Yes" : "No");
}
1;
```

SECTION 4

Standard plugins

The FIFO server provides two standard plugins as part of the EMu 4.0.01 distribution. These are:

- *Centroid*
- *System Lookup*

Centroid

The *Centroid* plugin returns a single latitude / longitude point representing the centre of a set of latitude / longitude points. The points can be supplied in either DMS (Degrees Minutes Seconds) format or as a decimal degree. Each point must be preceded by either DMS or DEC depending on the type of points supplied. The value returned is the centroid in both DMS and DEC formats. The precision of the supplied points is maintained in the result.

```
emufifo Centroid << EOF
DMS|14 45 12 N|43 15 W
DEC|14.7845|-43.27
EOF
14 46 8.1 N|43 15 36 W|14.76892|-43.260
```

System Lookup

The *System Lookup* plugin returns the text value for a given system lookup list. The name of the lookup list is supplied as input and the language dependent value is returned.

```
emufifo "System Lookup" "System Yes"
Yes
```

Index

C

C++ Client Code • 6

Centroid • 13

F

FIFO Server • 1

FIFO Server plugins • 7

I

Invoking the FIFO server • 3

K

KE Texpress Validation • 5

O

Overview • 1

S

Scripts and command line • 4

Standard plugins • 13

System Lookup • 13