EMu Documentation

Understanding the KE IMu Server

Document Version 1.0

EMu Version 4.0

www.kesoftware.com

Contents

SECTION 1

Introduction

This document assumes that readers understand object-oriented Perl and have read the relevant *Using KE IMu API* document (PHP, C#, etc.)

SECTION 2

How a Handler Works

Code developed using the IMu client libraries connects to an IMu server and then creates handler objects which submit requests to the server and receive responses. The distinguishing feature of a handler is that a corresponding object is created in the IMu server for each one of them in order to service its requests.

The basic functionality of a handler is contained in IMu's `Handler` class. Commonly used handlers, such as `Module` and `Cursor`, are implemented as subclasses of `Handler` and consequently provide higher-level methods which hide some of the operation of the handler.

Using the Handler class

It is possible to create an instance of a `Handler` directly and to use it to communicate with a corresponding server-side object.

The general principle of using a handler in this way is straightforward:

1. Create a handler and, optionally, tie it to an existing session.
   If a handler is not tied to a session, the IMu client library will create a new session, using the default.
2. Next, specify the name of the server-side object to be created when the handler communicates with the server.
3. Finally, call the handler's `call` method, passing it the name of a method in the server-side object to be invoked and a set of parameters to be passed to the method.

By way of example, the standard IMu server installation includes a set of handlers used for testing. One of these is `Test::Convert`. This includes a `toLower` method which takes a single string and converts it to its lower case equivalent. The following code illustrates how to create a `Test::Convert` handler and call its `toLower` method:

PHP

```
/* Include the handler code */
require IMu::$lib . '/handler.php';
…
/* Connect to a server */
$session = new IMuSession('server.com', 12345);
$session->connect();

/* Create a new handler object */
$handler = new IMuHandler($session);

/* Name the server-side object to be created */
$handler->name = 'Test::Convert';

/* Call the toLower method */
$result = $handler->call('toLower', 'miXedCaSE');
```

C#

```
using System;
…
/* Connect to a server */
IMu.Session session = new IMu.Session("server.com", 12345);
session.Connect();

/* Create new handler object */
IMu.Handler handler = new IMu.Handler(session);

/* Name the server-side object to be created */
handler.Name = "Test::Convert";

/* Call the toLower method */
Object result = handler.Call("toLower", "miXedCaSE");
```

Communication with the Server

What happens when this code is executed?

Once the connection to the server has been established (see *Using KE IMu API* for details), the `Handler` object is created and the name of the server-side class to be created is set in the `name` property. This is straightforward client-side code; there is no communication with the server up until this point.

It gets more interesting when the `Handler` object's `call` method is invoked. First, the `call` method puts its two arguments into an associative array. In our example this creates an associative array which contains the following name / value pairs:

```
"method" => "toLower"
"params" => "miXedCaSE"
```

This array is then passed to `Handler`'s `request` method, which adds further elements to the associative array. In this case the name of the server-side class to be created is added:

```
"method" => "toLower"
"params" => "miXedCaSE"
"name" => "Test::Convert"
```

This array is then passed to the `Session` object's `request` method. This method may add further entries to the associative array but its main role is to submit the request (in the form of the associative array) to the server and receive a response.

Sending the request to the `Session` object serialises it. The request is serialised using JavaScript Object Notation (JSON).

JSON is a lightweight but flexible data-interchange format. More information is available at http://www.json.org.
The format used has one addition to standard JSON, allowing binary objects to be transmitted as raw bytes rather than encoding them as JSON strings. This can save a significant amount of processing when transmitting large binary objects such as images and videos.

The associative array passed to the server is serialised as a JSON object:

```
{
 "method" : "toLower",
 "params" : "miXedCaSE",
 "name" : "Test::Convert"
}
```

Once the request has been serialised, the `Session` object passes it to the server.

Handler Names and Packages

The server receives the JSON request, unserialises it into a Perl hash and processes it. The first step in the processing is to check the request for a `name` element. If present, the server interprets this as a request to create a new server-side handler, which is a Perl package. To generate the name of the required Perl package the server adds the text `KE::Server::Handler` before the value in the `name` element.

For example, in our previous code, we requested a handler named `Test::Convert`. The package name generated by the server is therefore `KE::Server::Handler::Test::Convert`.

The server adds the `KE::Server::Handler` prefix to the handler name for security reasons: to prevent the client code from requesting to load any arbitrary Perl package.

The server then tries to load the package, first from the back-end environment's `local/etc/imuserver` directory. If the package is not found there, the server will try to load the package from the environment's `etc/imuserver` directory. Loading packages in this way provides a way for local handlers to override standard ones.

The server uses Perl's standard `use` mechanism to locate and load the packages. At start-up the server adds the two directories (`local/etc/imuserver` and `etc/imuserver`) to the front of the `@INC` array. This ensures that these two directories are checked first when loading a package.
However, if the package is not found in either of these directories Perl will use the rest of the `@INC` array to try to locate the package. The `@INC` array will contain the set of directories specified in the back-end environment's `PERL5LIB` environment variable and the standard set of Perl system directories.
More information regarding Perl's `use` mechanism is available at http://perldoc.perl.org/functions/use.html.

In our example, the server has been requested to load the `KE::Server::Handler::Test::Convert` package. To achieve this the server will first try to load the file:

`local/etc/imuserver/KE/Server/Handler/Test/Convert.pm`

If that file cannot be loaded, the server will then try to load:

`etc/imuserver/KE/Server/Handler/Test/Convert.pm`

If the package cannot be loaded, the server will return an error response. See Server Errors (page 14) for more information about errors.

Handler Processing

When the package has been loaded the server creates a new package object and uses this object to service the request: the server calls the requested method and passes it any parameters that were passed as part of the request.

To better understand how the server-side handler operates we will look at the code for `Test::Convert`.

```
use strict;
use warnings;

package KE::Server::Handler::Test::Convert;

use base 'KE::Server::Handler';
…
sub method_toLower
{
 my $this = shift;
 my $value = shift;
 return lc($value);
}
…
1;
```

The handler includes Perl's standard `strict` and `warnings` directives. All handlers should begin in this way.

As explained earlier (page 9), all handlers are Perl packages. The name of the package must begin with `KE::Server::Handler`. The package in this example is declared to be `KE::Server::Handler::Test::Convert`.

All handlers must be subclasses of the `KE::Server::Handler` package. The Perl `base` directive is used to specify the handler's immediate base class. This directive saves us having to `use` the base class' package and setup Perl's magic `@ISA` array. In this case our handler is an immediate subclass of `KE::Server::Handler`.

Each method that can be called from the client is implemented as Perl sub. The sub name must start with `method_`. The rest of the sub's name is the name of the method as it is called from the client. In our example, the method that is called as `toLower` in the client is implemented as a sub `method_toLower`.

The server forces methods which can be called from client-side code to start with `method_`. This ensures that the client cannot call an arbitrary sub in the package.

By convention, methods are named using so-called "camel case". Method names being with a lower case letter but any subsequent words start with an upper case letter: hence the name `toLower`.

The arguments passed to the method are straightforward. As with all Perl object-oriented code, the first argument passed is a reference to the Perl "object" itself (technically a blessed scalar reference). The second argument is the value passed in the `params` element from the client. In the example above it is a simple string.

The second argument to the `call` method (and hence the second argument passed to the server-side method) can be an associative array. This is useful when passing several pieces of information.

For example, the `Test::Convert` handler has another method, called `convert`, which accepts an `action` flag and a `value` string. The `action` flag can be the word `lower` or the word `upper` and the value string is converted appropriately. These parameters are passed to the server using an associative array:

PHP

```
$handler = new IMuHandler($session);
$handler->name = 'Test::Convert';
$paramaters = array();
$parameters['action'] = 'lower';
$parameters['value'] = 'miXedCaSE';
$result = $handler->call('convert', $parameters);
```

C#

```
IMu.Handler handler = new IMu.Handler(session);
handler.Name = "Test::Convert";
Hashtable parameters = new Hashtable();
parameters.Add("action", "lower");
parameters.Add("value", "miXedCaSE");
Object result = handler.Call("convert", parameters);
```

The server method uses the elements of the array as appropriate:

```
sub method_convert
{
 my $this = shift;
 my $params = shift;
 my $action = $params->{action};
 my $value = $params->{value};
 if (! defined($action))
 {
    return $value;
 }
 if (! defined($value))
 {
    return $value;
 }
 if ($action eq 'lower')
 {
    return lc($value);
 }
 elsif ($action eq 'upper')
 {
    return uc($value);
 }
 return $value;
}
```

This effectively passes a set of named arguments to a server-side method.

Server Response

Once the method has been called, the server returns a response to the client. The response is sent as a JSON object. The object includes a `status` element. This element contains the value `ok` if the request was processed correctly, or the value `error` if an error occurred during the processing.

If the `status` of the request is `ok`, the response object will also contain a `result` element. This contains the value actually returned by the server-side method. For example, the server's response to the `toLower` request will look like this:

```
{
 "status" : "ok",
 "result" : "mixedcase"
}
```

The client-side `Session` object receives this response and processes it. First it checks the `status` element. If the `status` value is `ok`, the response is returned to the `Handler` object's `request` method. This method checks for further elements in the response before returning the response to the `call` method, which finally returns the value in the `result` element. See *Re-using a Handler* (page 15) for details.

For example, the following code will print out the text `mixedcase`:

PHP

```
$result = $handler->call('toLower', 'miXedCaSE');
print("$result\n");
```

C#

```
Object result = handler.Call("toLower", "miXedCaSE");
System.Console.WriteLine(result);
```

The result need not be a simple value. It may be an associative array or a list as well.

Server Errors

If an error occurs while the server is processing a request, it will return a `status` value of `error`. If this happens, the client's `Session` request method will not return but will instead throw an exception. The type of exception will be an IMu `Exception` class. The `Exception` will contain the server-side error identifier in its `id` member. The error message (in the appropriate language) will be returned by the `Exception`'s `getString` method. For example:

PHP

```
try
{
 …
 $result = $handler->call('toLower', 'miXedCaSE');
 print("$result\n");
 …
}
catch (IMuException $error)
{
 print("An error occurred: " . $error->getString());
}
```

C#

```
try
{
 …
 Object result = handler.Call("toLower", "miXedCaSE");
 System.Console.WriteLine(result);
 …
}
catch (IMu.Exception error)
{
 System.Console.WriteLine("An error occurred: " +
     error.GetString());
}
```

Re-using a Handler

By default, the server will destroy a handler when one of its methods has been called. Client code can override this behaviour by setting the `Handler`'s `destroy` member to `false` (see *Using KE IMu API* for details). When the client subsequently calls a server-side method, the `destroy` value is passed as part of the request.

For example:

PHP

```
$handler = new IMuHandler();
$handler->name = 'Test::Convert';
$handler->destroy = false;
$handler->call('toLower', 'miXedCaSE');
```

C#

```
IMu.Handler = new IMu.Handler();
handler.Name = "Text::Convert";
handler.Destroy = false;
handler.Call("toLower", "miXedCaSE");
```

will result in the following name / value pairs being sent to the server:

```
"method" => "toLower"

"params" => "miXedCaSE"

"name" => "Test::Convert"

"destroy" => "false"
```

When the server processes this request it will not destroy the handler once the method has been called. Instead it will allocate the handler a unique identifier and return the identifier in the response:

```
{
    "status" : "ok",
    "result" : "mixedcase",
    "id" : "4c37"
}
```

The `Handler`'s `request` method (which receives the server's response from the `Session`'s `request` method) stores the identifier in the handler object. It then uses this identifier in any subsequent requests made by the handler. Consider the following example:

PHP

```
$handler = new IMuHandler();
$handler->name = 'Test::Convert';
$handler->destroy = false;
$handler->call('toLower', 'miXedCaSE');
// second call to same handler
$result = $handler->call('toUpper', 'all lower');
```

C#

```
IMu.Handler = new IMu.Handler();
handler.Name = "Test::Convert";
handler.Destroy = false;
handler.Call("toLower", "miXedCaSE");
// second call to same handler
handler.Call("toUpper", "all lower");
```

The second `call` will result in the request sent to the server containing the following name / value pairs:

`"method" => "toUpper"`

`"params" => "all lower"`

`"id" => "4c37"`

`"destroy" => "false"`

When the server receives this request it will not try to create a new handler (which would be problematic anyway as there is no `name` element passed in the request). Instead it will use the existing handler whose identifier is `4c37`. If no handler with the correct identifier is found, the server will return an error response.

Reconnecting to a Server

When working in a stateless environment such as a web server, IMu client code often needs to reconnect to the same handler. To do this the client not only needs to specify the identifier of a handler it wants to re-use, but also needs to ensure that it connects to the same server process (see *Using KE IMu API* for details).

To do this, the client code sets the `Session` object's `connection` member to `suspend`. When the next request is made, the `Session`'s `request` method adds the setting:

```
"connection" => "suspend"
```

to the request. When the server process handles this request, it starts to listen for connections on a second port, one that is unique to that process. The server then tells the client the number of the port to reconnect on subsequently by including a reconnect member in the JSON object returned as the response:

```
{
    …
    "reconnect" : 45679,
    …
}
```

The `Session`'s `request` method stores the value in its `port` member.

SECTION 3

Creating a New Handler

Creating a new handler is relatively straightforward once it is understood how a handler works. In this section we build a simple handler to illustrate how it is done. The handler will check the status of the EMu background loads.

The core of a handler is its server-side Perl package. The package must be a subclass of the `KE::Server::Handler` package and the package's full name must begin with `KE::Server::Handler`. Our handler will be called `Example` so its package name will be `KE::Server::Handler::Example`.

In order for the server to be able to load the package it will need to be stored under either the `local/etc/imuserver` or `etc/imuserver` directory. It is good practice to develop the package in the `local/etc/imuserver` directory. This means we will be creating a `local/etc/imuserver/KE/Server/Handler/Example.pm` file.

Basic Structure

Each handler should have a standard structure. The template for our `Example` handler is:

```
use strict;
use warnings;

package KE::Server::Handler::Example;

use base 'KE::Server::Handler';
…
1;
```

All Perl packages should include the `strict` and `warnings` directives.

All Perl packages must end with a value that evaluates to true so that the code loading the package can determine whether the package was loaded successfully. Using the value `1` as a statement on its own is the conventional way of doing this.

The code follows the requirements of naming and subclassing described above.

In this case it is not necessary to provide a constructor method (i.e. a Perl `sub new`). In more complicated handlers, however, it may be necessary to do so.

Adding a Method

Each method which is able to be called from the client must be implemented as a Perl `sub` whose name starts with:

`method_`

In our example we will start by providing a single method called `checkLoad`, which will take the name of a background load and return information about its status.

"camel case" is conventionally used for method names. Actually this is technically "lower camel case" where the first word starts with a lower case letter but subsequent words in the name begin with an upper case letter. Underscores should not be used in method names. Also by convention the methods are named to describe what they do and so, typically, start with a verb.

This means our Perl package must include a `sub` called `method_checkLoad`. The method must take two arguments, the object reference and the parameters passed to the `call` method in the client code. The template for our method looks like this:

```
sub method_checkLoad
{
 my $this = shift;
 my $loadName = shift;

 # check the load

 # return the result (just a place holder at the moment)
 return $loadName . ' is running??';
}
```

There are many ways of accessing the arguments passed to a Perl sub. The technique of copying each argument (using `shift`) into its own local variable, as shown above, is very common and recommended.

Trying It Out

We now have a new handler with a method that can be called. Provided we have installed the package's `.pm` file in the correct place, we should now be able to write a simple client-side test program.

First, restart the back-end server:

```
emuload restart imuserver
```

or, if that doesn't work:

```
emuload stop imuserver && emuload start imuserver
```

When a new package is installed it is not strictly necessary to restart the IMu server. However, if changes are made to a package that the server may have already loaded, it is necessary to restart the server to ensure that it loads the latest version. The simplest rule is always restart the server when testing new server-side handlers.

We can then write a simple client test program. The key parts of the test program are as follows:

PHP

```
$handler = new IMuHandler;
$handler->name = 'Example';
$result = $handler->call('checkLoad', 'audit');
print("$result\n");
```

C#

```
IMu.Handler handler = new IMu.Handler();
handler.Name = "Example";
Object result = handler.Call("checkLoad", "audit");
System.Console.WriteLine(result);
```

All being well, this program should produce the output:

```
audit is running??
```

If an error occurs, the `call` method will throw an exception.

Server Tracing

When the IMu server runs, its output and error streams are directed into a standard EMu load log file.

The current log file will be the most recently created file in the `loads/imuserver/logs` directory.

The server writes tracing information into this log file. The tracing information is typically date and time stamped and includes the `pid` of the server process which writes the information. Here is an example of the information written when our sample program is run:

```
2010-02-23 14:57:05: 14974 connection on port 45678
2010-02-23 14:57:05: 14974 listening...
2010-02-23 14:57:05: 14973 handling connection in this process
2010-02-23 14:57:05: 14975 listening...
2010-02-23 14:57:05: 14973 request:
{
    'params' => 'audit',
    'name' => 'Example',
    'method' => 'checkLoad'
}
2010-02-23 14:57:05: 14973 creating new Example handler
2010-02-23 14:57:05: 14973 destroying handler
2010-02-23 14:57:05: 14973 handler (KE::Server::Handler::Example) is being destroyed (garbage collection)
2010-02-23 14:57:05: 14973 response:
{
    'status' => 'ok',
    'result' => 'audit is running??'
}
2010-02-23 14:57:05: 14973 raising SocketEOF
2010-02-23 14:57:05: 14973 caught stream error: SocketEOF
2010-02-23 14:57:05: 14973 stream is being destroyed (garbage collection)
2010-02-23 14:57:05: 14973 listening...
```

In this trace we can see the arrival of our request, the creation of the new handler and the response. Notice also that the server destroys the handler once it has finished

servicing our request because we did not request otherwise.

Reviewing the trace output is a good way to check the operation of a handler.

A handler can add its own tracing information. To do so the package file must include `KE::Server::Common`. This module includes a global sub called `trace` which can be used to write information to the `trace` output. The `trace` sub takes at least two arguments. The first is the `trace` level (an integer). If this level is less than the level set for the server, the `trace` is written to the log. Otherwise it is discarded. The second argument is a simple `printf` style format string.

We can add tracing to our `Example` handler:

```
use KE::Server::Common;

sub method_checkLoad
{
 my $this = shift;
 my $loadName = shift;
 …
 trace(2, 'load to check is %s', $loadName);
 …
}
```

After restarting the server and re-running our test program, the log file includes our trace line:

```
2010-02-24 09:24:00: 1166 creating new Example handler
2010-02-24 09:24:00: 1166 load to check is audit
2010-02-24 09:24:00: 1166 destroying handler
```

Returning a Result

Our simple example needs to return whether the load is running or not. To do this it will return a status of `alive` or `dead`. Returning a value is straightforward. Here is our first implementation:

```
sub method_checkLoad
{
 my $this = shift;
 my $loadName = shift;

 trace(2, 'load to check is %s', $loadName);

 my $check = `emuload status "$loadName"`;
 my $status = 'dead';
 if ($check =~ /alive/)
 {
    $status = 'alive';
 }

 trace(2, '%s load is %s', $loadName, $status);

 return $status;
}
```

We can improve this a little. If the load is `alive`, we will also return the load's process id and the time it was started. The best way to return this information is as a Perl hash, which will be serialised as a JSON object:

```
sub method_checkLoad
{
 my $this = shift;
 my $loadName = shift;

 trace(2, 'load to check is %s', $loadName);

 my $check = `emuload status "$loadName"`;
 my $result = { status => 'dead' };
 if ($check =~ /(\d+)\s+alive\s+(.*)/)
 {
    $result->{status} = 'alive';
    $result->{pid} = $1;
    $result->{startTime} = $2;
 }

 trace(2, '%s load is %s', $loadName, $result->{status});

 return $result;
}
```

Handling Errors

Our example is not particularly robust. If no load name is passed by the client program, the `$loadName` variable will be undefined. Using an undefined value in certain contexts will cause Perl to generate warnings (which will appear in the log file). In this instance, using an undefined value or an empty string as the load name will actually cause the `emuload` command to return the status of all the EMu loads. This is not what our code expects and so odd results will be returned.

It is better to catch these cases and generate an error. To generate an error the code should call Perl's `die` sub and pass to it a `KE::Server::Exception` object. This exception will be caught by the server's request processing loop and automatically returned to the client.

To create an exception the handler code must include `KE::Server::Common`. This module includes a global sub called `raise`, which creates a `KE::Server::Exception` object from its arguments and calls `die`. The first argument to `raise` is a trace level (just as with `trace`). This is used to write information to the log about the exception being generated. The second argument is the exception identifier. This identifier is returned to the client, along with any further arguments passed to raise. The client `Session` object then raises its own exception:

```
sub method_checkLoad
{
 my $this = shift;
 my $loadName = shift;

 if (! defined($loadName) || $loadName eq '')
 {
        raise(2, 'ExampleMissingLoadName');
 }

 trace(2, 'load to check is %s', $loadName);
 …
}
```

Creating a Client Handler

Using simple handlers as in our example is straightforward using no more than the client's `Handler` class itself. However, for more complex handlers you may want to provide a client-side wrapper class that makes the handler easier to use.

A client-side handler must be a subclass of the `Handler` class. Typically the class will provide methods for each of the corresponding server-side methods. Here is an example of a wrapper for our simple `Example` handler:

PHP

```
require_once IMu::$lib . '/handler.php';

class Example extends IMuHandler
{
 public function
 __construct($session = false)
 {
     parent::__construct($session);
     $this->name = 'Example';
 }

 public function
 checkLoad($name)
 {
     return $this->call('checkLoad', $name);
 }
}
```

C#

```
class Example : IMu.Handler
{
 public
 Example(IMu.Session session)
     : base(session)
 {
     Name = "Example";
 }

 public
 Example()
     : base()
 {
     Name = "Example";
 }

 public Object
 CheckLoad(string name)
 {
     return Call("checkLoad", $name);
 }
}
```

As a result, the handler can be used in a simpler, more natural way:

PHP

```
$example = new Example;
$result = $example->checkLoad('audit');
```

C#

```
Example example = new Example();
Object result = example.CheckLoad("audit");
```

Another good reason to create a client handler class is to simplify dealing with the value returned from the server. The `Handler` class' `call` method returns a generic value. This is flexible but can be inconvenient to deal with.

Here we add a simple client-side `ExampleResult` class to make it easier to use the information returned by `checkLoad`:

PHP

```
class Example extends IMuHandler
{
 …
 public function
 checkLoad($name)
 {
     $array = $this->call('checkLoad', $name);
     $result = new ExampleResult;
     $result->alive = $array['status'] == 'alive';
     if (array_key_exists('pid', $array))
         $result->pid = $array['pid'];
     if (array_key_exists('startTime', $array))
         $result->startTime = strtotime($array['startTime']);
     return $result;
 }
}

class ExampleResult
{
 public $alive;
 public $pid;
 public $startTime;
}
```

C#

```
class Example : IMu.Handler
{
 …
 public ExampleResult
 CheckLoad(string name)
 {
     Hashtable hash = (Hashtable) Call("checkLoad", name);
     ExampleResult result = new ExampleResult();
     result.Alive = hash["status"].ToString().Equals("alive");
     if (hash.ContainsKey("pid"))
         result.Pid = int.Parse(hash["pid"].ToString());
     if (hash.ContainsKey("startTime"))
         result.StartTime =
            DateTime.Parse(hash["startTime"].ToString());
     return result;
 }
}

class ExampleResult
{
 public bool Alive;
 public int Pid;
 public DateTime StartTime;
}
```

This allows the handler to be used without knowledge of associative arrays and complex data structures:

PHP

```
$example = new Example;
$result = $example->checkLoad('audit');
if ($result->alive)
 print("The audit load is alive (pid = %d)\n", $result->pid);
else
 print("The audit load is dead\n");
```

C#

```
Example example = new Example();
ExampleResult result = example.CheckLoad("audit");
if (result.Alive)
 Console.WriteLine("The audit load is alive (pid = {0})",
         result.Pid);
else
 Console.WriteLine("The audit load is dead");
```

The Handler Environment

All server-side handlers are created by a `KE::Server::Listener` object. There is a `KE::Server::Listener` object for each server process which is running. This object is responsible for listening for new client connections, accepting these connections, creating new handlers, passing requests to handlers and destroying handlers when necessary.

Each handler contains a reference to the listener object which created it. The handler can use this reference to interact with the listener. The listener provides some methods that handlers may find useful. These are covered in the following sections.

Accessing EMu Tables

One of the most common tasks for a server-side handler is to access an EMu table. The server provides a `KE::Server::Module` package to provide this access.

 The `Module` handler makes extensive use of this package.

A handler should not try to create a `KE::Server::Module` object directly. Instead it asks the listener to create the object for it. The reason for this is that certain tables subclass `KE::Server::Module` to provide additional functionality. The listener object knows how to create the appropriate object for each table. In this sense the listener acts as factory.

To get a new table object, the handler should call the listener's `getModule` method, passing the name of the table:

```
my $module = $this->{listener}->getModule('eparties');
```

The module object provides low-level access to the EMu table. The following methods can be used:

- `findKey($key)`
  Searches the module for the record with a key of `$key`.
  Returns a `KE::Server::Module::Cursor` object.

 Module Cursors are explained below (page 32).

- `findKeys($keys)`
  Searches the module for the set of keys passed in the array reference `$keys`.
  Returns a `KE::Server::Module::Cursor` object.
- `findTerms($terms)`
  Searches the module for the set of terms passed in `$terms`. The terms should be specified in the same way as for the `findTerms` method in the client-side `Module` class.
  Returns a `KE::Server::Module::Cursor` object.
- `findWhere($where)`
  Searches using the Texql where clause passed in `$where`.
  Returns a `KE::Server::Module::Cursor` object.
- `emptyCursor()`
  Creates an empty cursor. This is useful if you plan to insert records.
  Returns a `KE::Server::Module::Cursor` object.
- `addColumnSet($name, $columns)`
  Associates a set of columns (in `$columns`) with a logical name (in `$name`) for later use. Similar to the same method in the client-side `Module` class.

Module Cursors

Several `KE::Server::Module` methods return a `KE::Server::Module::Cursor` object. This object is used for working with the set of matching results for a table.

The cursor object includes the following methods for changing the current row within the result set:

- `load($flag, $offset)`
  Sets the current row to a position specified by `$flag` and `$offset`. These arguments work identically to those in the client-side `Module` class `fetch` method.
- `get($rownum)`
  Sets the current row to a specific row number (row numbers start from 1).
- `next()`
- `prev()`
- `first()`
- `last()`
  Moves the current row as the method name suggests.

All these methods return `true` if the operation succeeded and `false` otherwise. This is particularly useful with the `next` and `prev` methods for moving through the result set.

These methods do not raise an exception if end-of-file is reached. They simply return `false`.

The cursor object also includes other methods:

- `tell()`
  Returns the current row number.
- `hits()`
  Returns the current estimated number of matches.
- `fetch($columns)`
  Retrieves the information for the set of columns passed in the array reference `$columns` from the current records.
  Returns the fetched columns as a Perl hash.
- `store($values)`
  Updates the current record. The `$values` argument is a reference to a Perl hash. The hash consists of a set of column name / value pairs to be updated in the current record.

- `insert($values)`
  Inserts a new record into the table. The `$values` argument is a reference to a Perl hash. The hash consists of a set of column name / value pairs to be inserted into the newly created record.
  Returns the irn of the newly created record.
- `sort($columns, $flags, $langid)`
  Sorts the result set. The `$columns` argument is a string specifying the columns by which to sort. The `$flags` argument is a reference to an array specifying flags controlling the behavior of the sort. The flags can be any of the following strings:
  - word-based
  - full-text
  - compress-spaces
  - case-sensitive
  - order-insensitive
  - null-low
  - extended-sort
  - table-as-text
  - report-array
  - report-xml
  - sort-text

  If the `report-xml` flag is included, the `sort` method returns a handle to a file containing the summary represented as XML. If the report-array flag is included, the method returns a reference to a perl array containing the summary.

Using the EMu Registry

The handler's listener object's `getModule` method can be used to get access to the EMu Registry table. However, the listener also provides a `getRegistry` method to do this. This method will only create a new instance of a `KE::Server::Module` for the eregistry table the first time it is called. Subsequent calls to `getRegistry` will return the same instance.

The object returned by the listener's `getRegistry` method is a `KE::Server::Module::eregistry` object. This is a subclass of the `KE::Server::Module` class and provides some useful methods for looking up Registry settings:

- `getValue($key, $default)`
  Looks up the key value specified in `$key`. The form of `$key` is:
  `Key|Key2|Key3…`
  For example, to look up a user joe's group, the key would be:
  `User|joe|Group`
  Returns the Registry value or `$default` if the entry is not found.
- `getSetting($key, $default)`
  Looks up a system setting. Calls `getValue` for
  `Group|Default|Setting|$key`
  and if that is not found, calls `getValue` for
  `System|Setting|$key`
  Returns the first setting found or `$default` if the entry is not found.
- `getMediaPath()`
  Determines the system media path list. First checks
  `System|Paths|ServerMediaPath`
  If this entry is not found, the method then checks
  `System|Paths|ServerPath`
  If this entry is found, the path is considered to be a single multimedia directory under the `ServerPath` value. If not found, the path is considered to be a single multimedia directory under the `$EMUPATH` environment variable.
  Returns a reference to an array containing the set of directories.
- `getMimeType($extension)`
  Looks up the mime type for the file extension passed in `$extension` by searching for
  `Mime|$extension|Content Type`
  Returns the mime type or `$default` if the extension is not found. If used in an array context, returns the two components of the mime type as separate array elements.

Temporary Files

The handler's listener object also provides a convenient way to manage temporary files. There are several methods that simplify using temporary files:

- `getTempPath()`
  Returns the Texpress `tmppath` setting. Usually `/tmp/texpress`.
- `getTempDir($template)`
  Creates a new directory inside the temporary directory. The `$template` argument is a pattern used to generate the name of the new directory. Any upper case `X` characters in the template are replaced by random characters until a new name has been generated. If no upper case characters are included in the template, `XXXX` is added to the end of the template. If no template is passed at all, the template `imuserverXXXX` is used.
  Returns the full path to the new directory.
- `getTempHandle($template)`
  Creates a new temporary file inside the temporary directory. The `$template` argument operates in the same way as for `getTempDir`. The file is opened for reading and writing. The file itself may be unlinked (removed from the file system) after it has been opened and will certainly be removed when the file is closed.
  Returns the handle to the new open file.
- `getTempFile($template)`
  Generates a unique name for a file to be created inside the temporary directory. The `$template` argument operates in the same way as for `getTempDir`. The file is not created. This makes it possible for another call to `getTempFile` with the same template to be allocated the same name. For this reason it is better to use `getTempHandle` if possible.
  Returns the full "unique" name.

Index