IMu Documentation

Using KE IMu's Perl API

Document Version 1

EMu Version 4.0

www.kesoftware.com

Contents

SECTION 1

Introduction

IMu, or Internet Museum, broadly describes KE Software's strategy and toolset for distributing data held within EMu via the Internet. Distribution includes the publishing of content on the web, but goes far beyond this to cover sharing of data via the Internet (portals, online partnerships, etc.); publishing content to new mobile technologies; iPod guided tours, etc.

To facilitate these various Internet projects, KE has produced a set of documents that describe how to implement and customize IMu components, including:

- APIs (for Developers)
- Web pages for publishing EMu
- Tools, including:
  - iPhone / mobile interfaces
  - iPod guided tours

This document describes use of the IMu Perl API.

SECTION 2

Using IMu's Perl library

The IMu Perl API source code bundle for version 1.0.03 (or higher) is required to develop an IMu-based application. This bundle contains all the modules that make up the IMu Perl API.

For building Perl applications the source code bundle must be extracted on the server machine and the path to the source code specified in the PERL5LIB environment variable or as an argument to the 'lib' module.

For example, if the IMu API source code is installed in the directory:

`/usr/local/lib/imu-1.0.03`

the following lines would be added to the Perl code:

```
use lib '/usr/local/lib/imu-1.0.03';
use IMu;
```

`IMu.pm` defines an IMu class. This class includes a member which contains information about the IMu installation. The class includes:

- `IMu::VERSION` - the version of this IMu release.

Test page

Compiling and running this very simple terminal based program is a good test of whether the development environment has been set up properly for using IMu:

```
#!/usr/bin/perl

use strict;
use warnings;
use lib '…/imu-1.0.03';
use IMu;

printf("IMu version %s\n", $IMu::VERSION);
exit(0);
```

Exceptions

Many of the methods in the IMu library objects throw exceptions (*die* in Perl terminology) when an error occurs. For this reason, code that uses IMu library objects should be surrounded with an `eval` block.

The following code is a basic template for writing Perl programs which use the IMu library:

```
…
eval
{
  # Create and use IMu objects
  …
};
if ($@)
{
  # Handle or report error
  …
}
```

Most IMu exceptions throw an `IMu::Exception` object. In many cases your code can simply handle the `$@` variable (as in this template). If more information is required about the `IMu::Exception` object thrown, see *Exceptions* (page 53).

Many of the examples that follow assume that code fragments have been surrounded with code structured in this way.

SECTION 3

Connecting to an IMu server

Most IMu based programs begin by creating a connection to an IMu server. Connections to a server are created and managed using IMu's `IMu::Session` module. Before connecting, both the name of the host and the port number to connect on must be specified. This can be done in one of three ways.

The simplest way to create a connection to an IMu server is to pass the host name and port number to the `IMu::Session` constructor and then call the `connect` method. For example:

```
use IMu::Session;
…
my $session = IMu::Session->new('server.com', 12345);
$session->connect();
```

Alternatively, pass no values to the constructor and then set the `host` and `port` properties (either by assigning to them directly or by using `setHost` and `setPort`) before calling `connect`:

```
use IMu::Session;
…
my $session = IMu::Session->new();

$session->{host} = 'server.com';
# or, equivalently
# $session->setHost('server.com');

$session->{port} = 12345;
# or, equivalently
# $session->setPort(12345);

$session->connect();
```

If either the `host` or `port` is not set, the `IMu::Session` class default value will be used. These defaults can be overridden by setting the class (static) properties `_defaultHost` and `_defaultPort`:

```
use IMu::Session;
…
IMu::Session::setDefaultHost('server.com');
IMu::Session::setDefaultPort(12345);
my $session = IMu::Session->new();
$session->connect();
```

This technique is useful when planning to create several connections to the same server or when wanting to get a handler object to create the connection automatically.

Handlers

Once a connection to an IMu server has been established, it is possible to create handler objects to submit requests to the server and receive responses.

When a handler object is created, a corresponding object is created by the IMu server to service the handler's requests.

All handlers are subclasses of IMu's `IMu::Handler` module.

You do not typically create an `IMu::Handler` object directly but instead use a subclass.

In this document we examine the most frequently used handler, `IMu::Module`, which allows you to find and retrieve records from a single EMu module.

SECTION 4

Accessing an EMu module

A program accesses an EMu module (or table, the terms are used interchangeably) using an `IMu::Module` class. The name of the table to be accessed is passed to the `IMu::Module` constructor. For example:

```
use IMu::Module;
…
my $parties = IMu::Module->new('eparties', $session);
```

This code assumes that an `IMu::Session` object called `$session` has already been created. If an `IMu::Session` object is not passed to the `IMu::Module` constructor, a session will be created automatically using the `_defaultHost` and `_defaultPort` class properties. See *Connecting to an IMu Server* (page 7) for details.

Once an `IMu::Module` object has been created, it can be used to search the specified module and retrieve records.

Searching a module

One of the following methods can be used to search for records within a module:

- `findKey`
- `findKeys`
- `findTerms`
- `findWhere`

findKey

The `findKey` method searches for a single record by its key.

For example, the following code searches for a record with a key of `42` in the Parties module:

```
use IMu::Module;
…
my $parties = IMu::Module->new('eparties', $session);
my $hits = $parties->findKey(42);
```

The method returns the number of matches found, which is either `1` if the record exists or `0` if it does not.

findKeys

The `findKeys` method searches for a set of key values. The keys are passed as a reference to an array:

```
my $parties = IMu::Module->new('eparties', $session);
my $keys = [52, 42, 17];
my $hits = $parties->findKeys($keys);
```

The method returns the number of records found.

findTerms

The `findTerms` method is the most flexible and powerful way to search for records within a module. It can be used to run simple single term queries or complex multi-term searches.

The terms are specified using an array reference. Each term is itself an array reference comprising two or three elements:

1. The first element contains the name of the column or an alias in the module to be searched.
2. The second element contains the value to search for.
3. A comparison operator can be included as a third element (see example 3 below).

   The operator specifies how the value supplied as the second argument of the array should be matched. In most cases, operators are the same as those used in TexQL (see KE's TexQL documentation for details).

   Specifying an operator is optional. If none is supplied, the operator defaults to `matches`. This is not a real TexQL operator, but is translated by the search engine as the most "natural" operator for the type of column being searched. For example, with text columns `matches` is translated as "contains" and with integer columns it is translated as "=".

Unless it is really necessary to specify an operator, consider using the `matches` operator, or better still supplying no operator at all as this allows the server to determine the best type of search.

The first element of each term may be the name of a search alias. A search alias associates a name with one or more actual columns. Aliases are created using the `addSearchAlias` or `addSearchAliases` methods.

Examples

1. To search for the name `Smith` in the *Last Name* field of the Parties module, the following term can be used:

```
my $search = ['NamLast', 'Smith'];
```

2. Specifying search terms for other types of columns is straightforward. For example, to search for records inserted on April 4, 2011:

```
my $search = ['AdmDateInserted', 'Apr 4 2011'];
```

3. To search for records inserted before April 4, 2011, it is necessary to add an operator:

```
my $search = ['AdmDateInserted', 'Apr 4 2011', '<'];
```

4. To specify more than one search term create a Boolean `AND` or `OR` term. This means that to find records which match both a *First Name* containing `John` and a *Last Name* containing `Smith` a reference to a terms array can be created as follows:

```
my $search =
[
  'and',
  [
    ['NamFirst', 'John'],
    ['NamLast', 'Smith']
  ]
];
```

or, equivalently,

```
my $terms = [];
push(@$terms, ['NamFirst', 'John']);
push(@$terms, ['NamFirst', 'Smith']);
my $search = ['and', $terms];
```

5. A set of terms where the relationship between the terms is a Boolean `OR` can be created just as simply. This means that:

```
my $search =
[
  'or',
  [
    ['NamFirst', 'John'],
    ['NamLast', 'Smith']
  ]
];
```

or, equivalently,

```
my $terms = [];
push(@$terms, ['NamFirst', 'John']);
push(@$terms, ['NamLast', 'Smith']);
my $search = ['or', $terms];
```

specifies a search for records where either the *First Name* contains `John` or the *Last Name* contains `Smith`.

6. Combinations of `AND` and `OR` search terms can be created simply by creating a nested array reference. To restrict the search for a *First Name* of `John` and a *Last Name* of `Smith` to matching records inserted before April 4, 2011 or on May 1, 2011, specify:

```
my $search =
[
   'and',
   [
      ['NamFirst', 'John'],
      ['NamLast', 'Smith'],
      [
         'or',
         [
            ['AdmDateInserted', 'Apr 4 2011', '<'],
            ['AdmDateInserted', 'Mar 1 2011']
         ]
      ]
   ]
];
```

7. To run a search, pass the terms array reference to the `findTerms` method:

```
my $parties = IMu::Module->new('eparties', $session);
my $search = ['NamLast', 'Smith'];
my $hits = $parties->findTerms($search);
```

   As with other `find` methods, the return value contains the estimated number of matches.

8. To use a search alias, call the `addSearchAlias` method to associate the alias with one or more real column names before calling `findTerms`. Suppose we want to allow a user to search the Catalog module for keywords. Our definition of a keywords search is to search the *SummaryData*, *CatSubjects_tab* and *NotNotes* columns. We could do this by building an `OR` search:

```
my $keyword = …;

my $terms =
[
   'or',
   ['SummaryData', $keyword],
   ['CatSubjects_tab', $keyword],
   ['NotNotes', $keyword]
];
```

Another way of doing this is to register the association between the name `keywords` and the three columns we are interested in and then pass the name `keywords` as the column to be searched:

```
my $keyword = …;
…
my $catalogue = IMu::Module->new('ecatalogue', $session);
my $columns =
[
   'SummaryData',
   'CatSubjects_tab',
   'NotNotes'
];
$catalogue->addSearchAlias('keywords', $columns);
…
my $search = ['keywords', $keyword];
$catalogue->findTerms($search);
```

An alternative to passing the columns as a reference to an array of strings is to pass a single string, with the column names separated by semi-colons:

```
my $keyword = …;
…
my $catalogue = IMu::Module->new('ecatalogue', $session);
my $columns = 'SummaryData;CatSubjects_tab;NotNotes';
$catalogue->addSearchAlias('keywords', $columns);
…
$search = ['keywords', $keyword];
$catalogue->findTerms($search);
```

The advantage of using a search alias is that once the alias is registered, a simple name can be used to specify a more complex `OR` search.

9. To add more than one alias at a time, use a hash of names and columns and call the `addSearchAliases` method:

```
my $aliases =
[
   'keywords' =>'SummaryData;CatSubjects_tab;'NotNotes',
   'title' => ['SummaryData', 'TitMainTitle']
];
$module->addSearchAliases($aliases);
```

findWhere

With the `findWhere` method it is possible to submit a complete TexQL `where` clause.

```
my $parties = IMu::Module->new('eparties', $session);
my $where = "NamLast contains 'Smith'";
my $hits = $parties->findWhere($where);
```

Although this method provides complete control over exactly how a search is run, it is generally better to use `findTerms` to submit a search rather than building a `where` clause. There are (at least) two reasons to prefer `findTerms` over `findWhere`:

1. Building the `where` clause requires the code to have detailed knowledge of the data type and structure of each column. The `findTerms` method leaves this task to the server. For example, specifying the term to search for a particular value in a nested table is straightforward. To find Parties records where the *Roles* nested table contains `Artist`, `findTerms` simply requires:
   `['NamRoles_tab', 'Artist']`
   On the other hand, the equivalent TexQL clause is:
   `exists(NamRoles_tab where NamRoles contains 'Artist')`
   The TexQL for double nested tables is even more complex.
2. More importantly, `findTerms` is more secure.
   With `findTerms` a set of terms is submitted to the server which then builds the TexQL `where` clause. This makes it much easier for the server to check for terms which may contain SQL-injection style attacks and to avoid them.
   If your code builds a `where` clause from user entered data so it can be run using `findWhere`, it is much more difficult, if not impossible, for the server to check and avoid SQL-injection. The responsibility for checking for SQL-injection becomes yours.

Number of matches

All the `find` methods return the number of matches found by the search. For `findKey` and `findKeys` this number is always the exact number of matches found. The number returned by `findTerms` and `findWhere` is best thought of as an estimate. This estimate is almost always correct but because of the nature of the indexing used by the server's data engine (Texpress) the number can sometimes be an over-estimate of the real number of matches. This is similar to the estimated number of hits returned by a Google search.

Getting information from matching records

`IMu::Module`'s `fetch` method is used to get information from the matching records once the search of a module has been run. The server maintains the set of matching records in a list and `fetch` can be used to retrieve any information from any contiguous block of records in the list.

The simplest form of the `fetch` method takes four arguments:

- `$flag`
- `$offset`
- `$count`
- `$columns`

$flag and $offset

The `$flag` and `$offset` arguments define the starting position of the block records to be fetched. The `$flag` argument is a string and must be one of:

- `'start'`
- `'current'`
- `'end'`

The `'start'` and `'end'` flags refer to the first record and the last record in the matching set. The `'current'` flag refers to the position of the last record fetched by the previous call to `fetch`. If `fetch` has not been called, `'current'` refers to the first record in the matching set.

The `$offset` argument is an integer. It adjusts the starting position relative to the `$flag`. A positive value for `$offset` specifies a start after the position specified by `$flag` and a negative value specifies a start before the position specified by `$flag`.

For example, calling `fetch` with a `$flag` of `'start'` and `$offset` of `3` will cause `fetch` to return records starting from the fourth record in the matching set. Specifying a `$flag` of `'end'` and a `$offset` of `-8` will cause `fetch` to return records starting from the ninth last record in the matching set.

To retrieve the next record after the last returned by the previous `fetch`, you would pass a `$flag` of `'current'` and a `$offset` of `1`.

$count

The `$count` argument specifies the maximum number of records to be retrieved.

Passing a `$count` value of `0` is valid. This causes `fetch` to change the current record without actually retrieving any data.

Using a negative value of `$count` is also valid. This causes `fetch` to return all the records in the matching set from the starting position (specified by `$flag` and `$offset`).

$columns

The `$columns` argument is used to specify which columns should be included in the returned records. The argument can be either a simple string or a reference to an array of strings. In its simplest form each string contains a single column name, or several column names separated by semi-colons or commas.

For example, to retrieve the information for both the *NamFirst* and *NamLast* columns, you would do one of:

```
my $parties = IMu::Module->new('eparties', $session);
my $columns = 'NamFirst;NamLast';
my $result = $parties->fetch('start', 0, 1, $columns);
```

-OR-

```
my $columns =
[
   'NamFirst',
   'NamLast'
];
my $result = $parties->fetch('start', 0, 1, $columns);
```

Return values

The `fetch` method returns records requested as a reference to a hash. This variable contains three members:

- `count` (an integer)
- `hits` (an integer)
- `rows` (a reference to an array)

The `count` property is the number of records returned by the `fetch` request.

The `hits` property is the estimated number of matches in the result set. This number is returned for each `fetch` because the estimate can decrease as records in the result set are processed by the `fetch` method.

The `rows` property is a reference to an array containing the set of records requested. Each element of the `rows` array is itself a reference to a hash. Each hash contains entries for each column requested.

The following example shows a simple search of the EMu Parties module using `findTerms` with `fetch` used to retrieve a set of records:

```
use IMu::Session;
use IMu::Module;

# perl standard library module for stringifying perl data structures
use Data::Dumper;
…
eval
{
   my $session = IMu::Session->new('server.com', 12345);
   my $parties = IMu::Module->new('eparties', $session);

   # Find all party records where Last Name contains 'Smith'
   my $search = ['NamLast', 'Smith'];
   my $hits = $parties->findTerms($search);

   # We want to fetch the irn, NamFirst and NamLast columns for each record.
   $columns =
   [
      'irn',
      'NamFirst',
      'NamLast'
   ];
   my $result = $parties->fetch('start', 0, 3, $columns);
   # display the result using a method from the Data::Dumper module
   print(Dumper($result));
};
if ($@)
{
…
}
```

The output of this code will be similar to:

```
$VAR1  =
     {
       'count' => 3,
       'hits' => 12,
       'rows' =>
         [
           {
             'rownum' => 1,
             'irn' => 722,
             'NamLast' => 'Chris',
             'NamFirst' => 'SMITH'
           },
           {
             'rownum' => 2,
             'irn' => 723,
             'NamLast' => 'Brad',
             'NamFirst' => 'Smith'
           },
           {
             'rownum' => 3,
             'irn' => 724,
             'NamLast' => 'Sylvia',
             'NamFirst' => 'Smith'
           },
         ]
     };
```

Notice that data for each row includes the `irn`, `NamFirst` and `NamLast` elements, which correspond to the columns requested. Also notice that a `rownum` element is included. This element contains the number of the record within the result set (starting from 1) and is always included in the retrieved records.

Nested tables are returned as a reference to an array of strings. For example, if a `$columns` argument of:

```
'NamLast;NamFirst;NamRoles_tab'
```

is passed, the variable returned will have a structure similar to:

```
$VAR1  =
     {
       'count' => 2,
       'hits' => 2,
       'rows' =>
         [
           {
             'rownum' => 1,
             'NamLast' => 'Ebb',
             'NamFirst' => 'Fred',
             'NamRoles_tab' =>
              [
                  'Lyricist',
                  'Pianist'
              ]
           },
         ]
     };
```

(Displayed using `Data::Dumper::Dumper`)

Attachments

The set of columns requested can be more than simple column names. Columns from modules which the current record attaches to can also be requested. For example, suppose that the Catalog module documents the creator of an object as an attachment (to a record in the Parties module) in a column called *CatCreatorRef*. If the Catalog module is searched, it is possible to get the creator's last name for each Catalog record in the result set as follows:

```
'CatCreatorRef.NamLast'
```

This technique can be extended to get information for more than one column:

```
'CatCreatorRef.(NamTitle;NamLast;NamFirst)'
```

The values are returned in a nested hash reference:

```
$VAR1  =
     {
       'count' => 1,
       'hits' => 1,
       'rows' =>
            [
              {
               'irn' => 5,
               'rownum' => 1,
               'CatCreatorRef' =>
                  {
                      'NamLast' => 'Mueck',
                      'NamTitle' => 'Mr',
                      'NamFirst' => 'Ron'
                  }
              }
            ]
     };
```

Users of the older EMuWeb system should note that it is possible to use an "arrow" (i.e. a hyphen followed by a greater-than sign) in place of the dot, e.g.:
`"CatCreatorRef->NamLast"`
Also note that it is not necessary to include the table name in the reference. For example:
`"CatCreatorRef->eparties->NamLast"`
is not necessary. The IMu server will accept this syntax and silently ignore the table name.

Reverse attachments

In addition to standard attachment columns, it is possible to request information from so-called reverse attachments. A reverse attachment refers to one or more records which attach to the current record.

For example, to retrieve information from a set of Catalog records which attach to the current Parties record via the Catalog's *CatCreatorRef* column, specify:

```
'<ecatalogue:CatCreatorRef>.(irn,TitMainTitle)'
```

The following code fragment retrieves Parties IRN `53` and displays the *CatCreatorRef* reverse attachments:

```
my $parties = IMu::Module->('eparties', $session);
my $hits = $parties->findKey(53);

my $columns =
[
   'irn',
   'NamFirst',
   'NamLast',
   '<ecatalogue:CatCreatorRef>.(irn,TitMainTitle)'
];

my $result = $parties->fetch('start', 0, 1, $columns);
print(Dumper($result));
```

The reverse attachments are returned as a reference to an array:

```
$VAR1  =
     {
       'count' => 1,
       'hits' => 1,
       'rows' =>
         [
           {
             'irn' => 53,
             'rownum' => 1,
             'NamLast' => ' Mueck ',
             'ecatalogue:CatCreatorRef' =>
              [
                  {
                      'irn' => 5,
                      'TitMainTitle' => 'In Bed'
                  },
                  {
                      'irn' => 50,
                      'TitMainTitle' => 'Man in Blankets'
                  },
              ],
              'NamFirst' => 'Ron'
           }
         ]
     };
```

Rename a column

It is possible to rename any column when it is returned by adding the new name in front of the real column being requested, followed by an equals sign.

For example, to request data from the *NamLast* column but rename it as `last_name`, specify:

```
'last_name=NamLast'
```

The returned `Map` will contain an element called `last_name` rather than `NamLast`.

This is particularly useful for complicated reverse attachment names:

```
'objects=<ecatalogue:CatCreatorRef>.(SummaryData)'
```

Grouping a set of nested table columns

A set of nested table columns can be grouped. Grouping allows the association between the columns to be reflected in the structure of the data returned. Consider the *Contributors* grid on the Details tab of the Narratives module, which contains two columns:

- *NarContributorRef_tab*
  which contains a set of attachments to records in the Parties module.
- *NarContributorRole_tab*
  which contains the roles for the corresponding contributors.

Each column can be retrieved separately as follows:

```
my $narratives = IMu::Module->new('enarratives', $session);

$narratives->findKey(2);

my $columns =
[
  'irn',
  'NarTitle',
  'NarContributorRef_tab.SummaryData',
  'NarContributorRole_tab'
];

my $result = $narratives->fetch('start', 0, 1, $columns);
print(Dumper($result));
```

This produces output such as:

```
$VAR1  =
    {
      'count' => 1,
      'hits' => 1,
      'rows' =>
        [
          {
            'NarTitle' => 'Portrait of William Wilberforce',
            'irn' => 2,
            'rownum' => 1,
            'NarContributorRole_tab' =>
             [
                 'Artist',
                 'Author'
             ],
            'NarContributorRef_tab' =>
             [
                 {
                     'SummaryData' => 'Rising, John'
                 },
                 {
                     'SummaryData' => 'Graham, Beverley'
                 }
             ]
          }
        ]
    };
```

Although this works fine, the relationship between the contributor and his or her role is unclear. Grouping can make the relationship far clearer.

To group the columns, surround them with square brackets:

```
'[NarContributorRef_tab.SummaryData,NarContributorRole_tab]'
```

With this single change, output of the previous code fragment looks like this:

```
$VAR1  =
     {
       'count' => 1,
       'hits' => 1,
       'rows' =>
         [
           {
             'NarTitle' => 'Portrait of William Wilberforce',
             'irn' => 2,
             'rownum' => 1,
             'group1' =>
              [
                  {
                      'NarContributorRole_tab' => 'Artist',
                      'NarContributorRef_tab' =>
                         {
                             'SummaryData' => 'Rising, John'
                         }
                  },
                  {
                      'NarContributorRole_tab' => 'Author',
                      'NarContributorRef_tab' =>
                         {
                             'SummaryData' => 'Graham, Beverley'
                         }
                  }
              ]
           }
         ]
     };
```

By default, the group is given a name of `group1`, `group2` and so on, which can be changed easily enough:

```
'contributors=[NarContributorRef_tab.SummaryData,
  NarContributorRole_tab]'
```

Column sets

Every time `fetch` is called and a set of columns to retrieve is passed, the IMu server must parse these columns and check them against the EMu schema. For complex column sets, particularly those involving several references or reverse references, this can take time.

If `fetch` will be called several times with the same set of columns, it is a good idea to register the set of columns once and then simply pass the name of the registered set each time `fetch` is called.

`IMuModule`'s `addFetchSet` method is used to register a set of columns. This method takes two arguments:

- The name of the column set.
- The set of columns to be associated with that name.

For example:

```
my $columns =
[
  'irn',
  'NamFirst',
  'NamLast'
];
$parties->addFetchSet('PersonDetails', $columns);
```

This registers the set of columns with the IMu server and gives it the name `PersonDetails`. This name can then be passed to any call to `fetch` and the same set of columns will be returned:

```
$parties->fetch('start', 0, 5, 'PersonDetails');
```

More than one set can be registered at once using `addFetchSets`. Simply build a hash containing each set:

```
my $sets =
{
  'PersonDetails' => ['irn', 'NamFirst', 'NamLast'],
  'OrganisationDetails' => ['irn', 'NamOrganisation']
};
$module->addFetchSets($sets);
```

Using column sets is very useful when maintaining state.

A simple example

In this example we build a simple command-line based Perl program to search the Parties module by *Last Name* and display the full set of results. The name to be searched for will be passed to the program as a command-line argument:

```
#!/usr/bin/perl

use strict;
use warnings;
use lib '…/imu-1.0.03';
use IMu::Session;
use IMu::Module;

if (! @ARGV)
{
   die("Usage: example name\n");
}
eval
{
   process($ARGV[0]);
};
if ($@)
{
   die("Sorry, an error occurred: $@\n");
}
exit(0);

sub process
{
   my $lastName = shift;

   my $session = IMu::Session->new('server.com', 12345);
   $session->connect();
   my $parties = IMu::Module->new('eparties', $session);

   # Build search term and run search
   my $terms = ['NamLast', $lastName];
   my $hits = $parties->findTerms($terms);

   # Build list of columns to fetch
   my $columns =
   [
      'NamFirst',
      'NamLast'
   ];

   # Fetch all the matches in one go by passing count < 0
   my $result = $parties->fetch('start', 0, -1, $columns);

   # Display the results
   print("Number of matches: $result->{hits}\n");
   my $rows = $result->{rows};
   foreach my $row (@$rows)
   {
```

```
    my $rownum = $row->{rownum};
    my $first = $row->{NamFirst};
    my $last = $row->{NamLast};
    print("$rownum: $first $last\n");
  }
}
```

The output generated looks like this:

```
Number of matches: 5
1: Percy JONES
2: Marilyn JONES
3: Lee Jones
4: David Jones
5: William Jones
```

Sorting

The matching set of results can be sorted using the `IMu::Module` `sort` method. This method takes two arguments:

- `$keys`
- `$flags`

$keys

The `$keys` argument is used to specify the columns by which to sort the result set. The argument can be either a simple string or a reference to an array of strings. Each string can be a simple column name or a set of column names, separated by semi-colons or commas. Each column name can be preceded by a + or –. A leading + indicates that the records should be sorted in ascending order. A leading – indicates that the records should be sorted in descending order.

For example, to sort a set of Parties records first by *Party Type* (ascending), then *Last Name* (descending) and then *First Name* (ascending):

```
my $keys = '+NamPartyType;-NamLast;+NamFirst';
```

-OR-

```
my $keys =
[
   '+NamPartyType',
   '-NamLast',
   '+NamFirst'
];
```

If a sort order (`+` or `-`) is not given, the sort order defaults to ascending.

$flags

The `$flags` argument is used to pass one or more flags to control the way the sort is carried out. As with the `$keys` argument, the `$flags` argument can be a simple string or a reference to an array of strings. Each string can be a single flag or a set of flags separated by semi-colons or commas.

The following flags control the type of comparisons used when sorting:

`'word-based'`	`sort` disregards all punctuation and white spaces (more than the one space between words). For example:
	`Traveler's Inn`
	will be sorted as
	`Travelers Inn`

`'full-text'`	`sort` includes all punctuation and white spaces. For example: `Traveler's Inn` will be sorted as `Traveler's Inn` and will therefore differ from: `Traveler's Inn`
`'compress-spaces'`	`sort` includes punctuation but disregards all white space (with the exception of a single space between words). For example: `Traveler's Inn` will be sorted as `Traveler's Inn`

If none of these flags is included, the comparison defaults to `'word-based'`.

The following flags modify the sorting behavior:	

`'case-sensitive'`	`sort` is sensitive to upper and lower case. For example:
	`Melbourne gallery`
	will be sorted separately to
	`Melbourne Gallery`

`'order-insensitive'`	Values in a multi-value field will be sorted alphabetically regardless of the order in which they display. For example, a record which has the following values in the *NamRoles_tab* column in this order:
	`Collection Manager`
	`Curator`
	`Internet Administrator`
	and another record which has the values in this order:
	`Internet Administrator`
	`Collection Manager`
	`Curator`
	will be sorted the same.

`'null-low'`	Records with empty fields will be placed at the start of the result set rather than at the end.
`'extended-sort'`	Values that include diacritics will be sorted separately to those that do not. For example, `entrée` will be sorted separately to `entree`.

The following flags can be used when generating a summary of the sorted records:

`'report'`	A summary of the sort is generated. The summary is contained in a reference to a hash. The result is hierarchically structured, summarizing the number of records which match each of the sort keys. See the example for an illustration of the structure.

`'table-as-text'`	All data from multi-valued columns will be treated as a single value (joined by line break characters) in the summary results hash reference.
	For example, for a record which has the following values in the *NamRoles_tab* column:
	`Collection Manager, Curator, Internet Administrator`
	the summary will include statistics for a single value:
	`Collection Manager`
	`Curator`
	`Internet Administrator`
	Thus the number of values in the summary results display will match the number of records.
	If this option is not included, each value in a multi-valued column will be treated as a distinct value in the summary. Thus there may be many more values in the summary results than there are records.

Return value

The `sort` method returns the undefined value unless the `report` flag is used.

If the `report` flag is used, the `sort` method returns a simple array reference representing a list of distinct terms associated with the primary key in the sorted result set.

Each element in the array is a hash. The hash contains three elements which describe the term:

- `value` (a string)
- `count` (an integer)
- `list` (an array reference)

The `value` element is the distinct value itself.

The `count` element is the number of records in the result set which have this value.

The `list` element is a nested array reference. This holds values for secondary sorts within the primary sort. This is illustrated in the following example:

Example

In this example we run a three-level sort on a set of Parties records, sorting first by *Party Type*, then *Last Name* (descending) and then by *First Name*. Setting up and running the sort is straightforward:

```
my $parties = IMu::Module->new('eparties', …);
…
$parties->findTerms(…);
…
my $keys =
[
   '+NamPartyType',
   '-NamLast',
   '+NamFirst'
];
my $flags =
[
   'full-text',
   'case-sensitive',
   'report'
];
my $result = $parties->sort($keys, $flags);
print(Dumper($result));
```

This will produce output similar to the following:

```
$VAR1  =
     [
     …
        {
          'count' => '2086',
          'value' => 'Person',
          'list' =>
            [
            …
               {
                'count' => '4',
                'value' => 'Young',
                'list' =>
                    [
                        {
                            'count' => '1',
                            'value' => 'Derek'
                        },
                        {
                            'count' => '1',
                            'value' => 'Don'
                        },
                        {
                            'count' => '1',
                            'value' => 'George'
                        },
                        {
                            'count' => '1',
                            'value' => 'Shirley'
                        },
                    ]
               },
            …
            ]
        }
     …
     ];
```

SECTION 5

Multimedia

The multimedia resources associated with an EMu record can be retrieved using the `IMu::Module fetch` method by specifying a special column called *multimedia*. When this column is requested the server returns the set of multimedia attachments associated with the record in question.

The set is returned as a reference to an array of hashes. Each hash includes the following information:

- `irn`
  The irn of the resource in EMu's Multimedia module.
- `type`
  The media type: typically `image`, `audio`, `video`, etc.
- `format`
  The media format or sub-type such as jpeg or tiff for image formats, wav or mpeg for audio.

This is equivalent to the column request:

```
multimedia=MulMultiMediaRef_tab.
(
  irn,
  type=MulMimeType,
  format=MulMimeFormat
)
```

with the addition that the result does not contain any empty entries (i.e. entries corresponding to null values in the *MulMultiMediaRef_tab* column) or any entries for Multimedia records which are not accessible via IMu.

For example:

```
my $session = IMu::Session->new('server.com', 12345);
$session->connect();
my $parties = IMu::Module->new('eparties', $session);

# Build search term and run search
my $terms = ['NamLast', 'Pavarotti'];
my $hits = $parties->findTerms($terms);

# Build list of columns to fetch
my $columns =
[
   'NamFirst',
   'NamLast',
   'multimedia'
];

# We are only interested in the first record
my $result = $parties->fetch('start', 0, 1, $columns);
my $row = $result->{rows}->[0];

# Display the results
my $first = $row->{NamFirst};
my $last = $row->{NamLast};
my $multimedia = $row->{multimedia};

print("First Name: $first\n");
print("Last Name: $last\n");
print('Multimedia: ', scalar(@$multimedia), "\n");
foreach my $entry (@$multimedia)
{
   my $irn = $entry->{irn};
   my $type = $entry->{type};
   my $format = $entry->{format};

   print("  irn $irn: $type/$format\n");
}
exit(0);
```

will produce output such as:

```
First Name: Luciano
Last Name: PAVAROTTI
Multimedia: 11
  irn 100096: image/gif
  irn 100100: image/gif
  irn 100101: image/gif
  irn 100102: image/gif
  irn 100105: image/jpeg
  irn 100095: video/quicktime
  irn 100103: video/quicktime
  irn 100098: audio/wav
  irn 100099: audio/wav
  irn 100104: audio/wav
  irn 100097: application/msword
```

The *multimedia* column is an example of an IMu "virtual" column. The column does not actually exist in the EMu table being accessed. Instead, the IMu server interprets the request for the column and builds an appropriate response. There are other virtual columns that can be used when accessing a record's multimedia attachments:

- *images*

  This returns the subset of multimedia attachments which have a mime type of `image`. Like *multimedia*, this is returned as a reference to an array of hashes for each image.
- *image*

  The preferred image from the set of images. Currently this is the same as the first entry in the array reference returned by images. However, future versions of EMu may allow another multimedia attachment to be flagged as the preferred image, in which case the *image* column will return information for the preferred resource, rather than the first one. This is returned as a reference to a hash.
- *videos*

  This returns the subset of multimedia attachments which have a mime type of `video`. Like *multimedia*, this is returned as a reference to an array of hashes for each image.
- *video*

  The preferred video from the set of videos. Currently this is the same as the first entry in the array reference returned by *videos*.

All these virtual columns act as reference columns into the Multimedia module. This means that other Multimedia columns can also be requested from the corresponding Multimedia record. For example, to include the publisher (*DetPublisher*) in the information returned for each attached multimedia resource:

```
multimedia.DetPublisher
```

The returned hashes will include a *DetPublisher* entry as well as the standard *irn*, *type* and *format* entries.

Any standard columns from the Multimedia module can be requested in this way. In addition, the Multimedia module includes a virtual column, *resource*, which can be used to get access to the contents of the actual multimedia resource. The *resource* column is returned as another hash. The object includes the following information:

- `identifier`

  The contents of the multimedia record's *MulIdentifier* field.
- `mimeType`

  The media type: typically `image`, `audio`, `video`, etc.
- `mimeFormat`

  The media format or sub-type such as jpeg or tiff for image formats, wav or mpeg for audio.
- `size`

  The size of the resource in bytes.
- `file`

  An open Perl file handle. This provides a read-only handle to a temporary copy of the resource itself. The temporary copy of the file is discarded when the file handle is closed or destroyed.

- `height`
  For images, the height of the image in pixels.
- `width`
  For images, the width of the image in pixels.

The following code fragment retrieves Parties IRN 53, displays the information for its preferred attached image and creates a copy of the resource in a file called `image-copy`:

```
my $parties = IMu::Module->new('eparties', $session);

# Build search term and run search
my $hits = $parties->findKey(53);

# Build list of columns to fetch
my $columns =
[
   'NamFirst',
   'NamLast',
   'image.resource'
];

# We are only interested in the first record
my $result = $parties->fetch('start', 0, 1, $columns);
my $row = $result->{rows}->[0];

my $image = $row->{image};
my $resource = $image->{resource};

# Print out information about the resource
print("identifier: $resource->{identifier}\n");
print("mimeType: $resource->{mimeType}\n");
print("mimeFormat: $resource->{mimeFormat}\n");
print("size: $resource->{size}\n");

# Save a copy of the resource
# Error checking for file open, read & writes ommited for brevity
my $temp = $resource->{file};
my $copy = IO::File->new('image-copy', '>');

my $buffer;
my $offset = 0;
for (;;)
{
   # read 4K at a time
   my $length = $temp->sysread($buffer, 4096, $offset);
   if ($length == 0)
   {
      last;
   }
   $copy->syswrite($buffer, $length, $offset);
   $offset += $length;
}
$copy->close();


exit(0);
```

This will produce output similar to:

```
identifier: LucianoPavarotti.gif
mimeType: image
mimeFormat: gif
size: 19931
```

as well as creating a file called `image-copy` which contains the copy of the image itself.

The previous example retrieves a binary copy of the master resource in its original format. It is also possible to modify how the resource is returned. This is done by adding modifiers to the *resource* column request. Modifiers are added after the column name and inside a set of braces.

The modifiers which can be applied to the *resource* column are:

- `encoding`

  Specifies that the resource returned should be encoded. The only currently supported encoding is `base64`. By default the resource is returned as raw binary data.

  Example:

  ```
  resource{encoding:base64}
  ```

- `checksum`

  Specifies that the information returned with the resource should include a checksum. The checksum requested can be `crc32` or `md5`.

  Example:

  ```
  resource{checksum:crc32}
  ```

In addition other modifiers can be applied to image resources:

- `format`

  Specifies the format of the required image. If the master image is already in the required format, then it is returned. Otherwise the image is reformatted on-the-fly and the reformatted image is returned.

  Example:

  ```
  resource{format:gif}
  ```

  This requests that the imaged is returned as a gif.

  The IMu server uses ImageMagick to process the image and the range of supported formats is very large. The complete list is available from:

  http://www.imagemagick.org/script/formats.php

- `height`

  Specifies the height of the image required in pixels. If the record contains a resolution with this height, this resolution is returned. Otherwise the closest matching larger resolution is resized to the requested height on-the-fly and the resized image is returned.

  Example:

  ```
  resource{height:200}
  ```

- `width`

  Specifies the width of the image required in pixels. If the record contains a resolution with this width, this resolution is returned. Otherwise the closest matching larger resolution is resized to the requested width on-the-fly and the resized image is returned.

  Example:

  ```
  resource{width:300}
  ```

- `bestfit`

  If set to `yes`, the image returned is the existing resolution which most closely matches the specified height or width. No on-the-fly resizing is done.

Example:

```
resource{height:300,bestfit:yes}
```

This returns the image closest to, but larger than, 300 pixels high.

- `aspectratio`

  Controls whether the image's aspect ratio should be maintained when both a height and a width are specified. If set to `no`, the aspect ratio is not maintained.

  Example:

  ```
  resource{height:300,width:300,aspectratio:no}
  ```

- `source`

  Controls which image is used as the basis for any reformatting that is required.

  By default, if no height or width is specified, the master is used as the source image. However, if a height or width is supplied, then by default the closest sized but larger resolution is used as the source. This saves processing time but may not produce the best result when dealing with lossy formats (such as jpeg). To override this, a source value of `master` can be specified.

  Example:

  ```
  resource{height:300,source:master}
  ```

  This specifies that the image is generated by resizing the master to 300 pixels high, rather than by using any appropriate resolution.

  The source value can also be `thumbnail`. In this case the image thumbnail is used as the source. Typically you would not want to apply size transformations to the thumbnail but this provides a simple way of retrieving the image's 90x90 thumbnail:

  ```
  resource{source:thumbnail}
  ```

SECTION 6

Maintaining state

One of the biggest drawbacks of the earlier example is that it fetches the full set of results at one time, which is impractical for large result sets. It is more practical to display a full set of results across multiple pages and allow the user to move forward or backward through the pages.

This is simple in a conventional application where a connection to the server is maintained until the user terminates the application. In a web implementation however, this seemingly simple requirement involves a considerably higher level of complexity due to the *stateless* nature of web pages. One such complexity is that each time a new page of results is displayed, the initial search for the records must be re-executed. This is inconvenient for the web programmer and potentially slow for the user.

The IMu server provides a solution to this. When a handler object is created, a corresponding object is created on the server to service the handler's request: this server-side object is allocated a unique identifier by the IMu server. When making a request for more information, the unique identifier can be used to connect a new handler to the same server-side object, with its state intact.

The following example illustrates the connection of a second, independently created `IMu::Module` object to the same server-side object:

```
# Create a module object as usual
my $first = IMu::Module->new('eparties', $session);

# Run a search - this will create a server-side object
my $keys = [1, 2, 3, 4, 5, 42];
$first->findKeys($keys);

# Get a set of results
my $result1 = $first->fetch('start', 0, 2, 'SummaryData');

# Create a second module object
my $second = IMu::Module->new('eparties', $session);

# Attach it to the same server-side object as the
# first module. This is the key step.
$second->{id} = $first->{id};

# Get a second set of results from the same search
my $result2 = $second->fetch('current', 1, 2, 'SummaryData');
```

Although two completely separate `IMu::Module` objects have been created, they are each connected to the same server-side object by virtue of having the same `id` property. This means that the second `fetch` call will access the same result set as the first `fetch`. Notice that a flag of `'current'` has been passed to the second call. The current state is maintained on the server-side object, so in this case the second call to `fetch` will return the third and

fourth records in the result set.

While this example illustrates the use of the `id` property, it is not particularly realistic as it is unlikely that two distinct objects which refer to the same server-side object would be required in the same piece of code. The need to re-connect to the same server-side object when generating another page of results is far more likely. This situation involves creating a server-side `IMu::Module` object (to search the module and deliver the first set of results) in one request and then re-connecting to the same server-side object (to fetch a second set of results) in a second request. As before, this is achieved by assigning the same identifier to the `id` property of the object in the second page, but two other things need to be considered.

First, by default the IMu server destroys all server-side objects when a session finishes. This means that unless the server is explicitly instructed not to do so, the server-side object may be destroyed when the connection from the first page is closed. Telling the server to maintain the server-side object only requires that the `destroy` property on the object is set to `0` (false) before calling any of its methods. In the example above, the server would be instructed not to destroy the object as follows:

```
my $module = IMu::Module->new('eparties', $session);
$module->{destroy} = 0;
# or, equivalently
# $module->setDestroy(0);
my $keys = [1, 2, 3, 4, 5, 42];
$module->findKeys($keys);
```

The second point is quite subtle. When a connection is established to a server, it is necessary to specify the port to connect to. Depending on how the server has been configured, there may be more than one server process listening for connections on this port. Your program has no control over which of these processes will actually accept the connection and handle requests. Normally this makes no difference, but when trying to maintain state by re-connecting to a pre-existing server-side object, it is a problem.

For example, suppose there are three separate server processes listening for connections. When the first request is executed it connects, effectively at random, to the first process. This process responds to the request, creates a server-side object, searches the Parties module for the terms provided and returns the first set of results. The server is told not to destroy the object and passes the server-side identifier to another page which fetches the next set of results from the same search.

The problem comes when the next page connects to the server again. When the connection is established any one of the three server processes may accept the connection. However, only the first process is maintaining the relevant server-side object. If the second or third process accepts the connection, the object will not be found.

The solution to this problem is relatively straightforward. Before the first request closes the connection to its server, it must notify the server that subsequent requests need to connect explicitly to that process. This is achieved by setting the `IMu::Session` object's `suspend` property to `1` (true) prior to submitting any request to the server:

```
my $session = IMu::Session->new('server.com', 12345);
my $module = IMu::Module->new('eparties', $session);
…
$session->{suspend} = 1;
# or, equivalently
# $session->setSuspend(1);

$module->findKeys(…);
```

The server handles a request to suspend a connection by starting to listen for connections on a second port. Unlike the primary port, this port is guaranteed to be used only by that particular server process. This means that a subsequent page can reconnect to a server on this second port and be guaranteed of connecting to the same server process. This in turn means that any saved server-side object will be accessible via its identifier. After the request has returned (in this example it was a call to `findKeys`), the `IMu::Session` object's `port` property holds the port number to reconnect to:

```
$session->setSuspend(1);
$module->findKeys(…);
$reconnect = $session->{port};
```

Example

This may seem a little complicated but it is not in fact too difficult to manage in practice.

To illustrate we'll modify the very simple results page of the earlier section (page 30) to display the list of matching names in blocks of five records per page. We'll provide simple **Next** and **Prev** links to allow the user to move through the results, and we will use some more `GET` parameters to pass the port we want to reconnect to, the identifier of the server-side object and the `rownum` of the first record to be displayed.

The code to be modified is in `results.pl` and is all inside the `eval` block (so we don't show the other code outside the `eval` block).

First, we create the `IMu::Session` object. We set the `port` property to a standard value unless a `port` parameter has been passed in the URL. We use a core Perl module `CGI` to handle getting parameters from the URL and outputing HTML elements:

```
use CGI;
my $cgi = CGI->new();

# Create new session object.
my $session = IMu::Session->new();
$session->setHost('server.com');

# Work out what port to connect to
my $port = 12345;
if (defined($cgi->param('port')))
{
  $port = $cgi->param('port')
}
$session->setPort($port);
```

Next we connect to the server. We immediately set the `suspend` property to `1` (true) to tell the server that we may want to connect again (this ensures the server listens on a new, unique port):

```
# Establish connection and tell the server we may want to
# re-connect
$session->connect();
$session->setSuspend(1);
```

We then create the client-side `IMu::Module` object and set its `destroy` property to `0` (false), ensuring the server will not destroy it:

```
# Create module object and tell the server not to destroy it.
my $module = IMu::Module->new('eparties', $session);
$module->setDestroy(0);
```

If the URL included a `name` parameter, we need to do a new search. Alternatively, if it included an `id` parameter, we need to connect to an existing server-side object:

```
# If name is supplied, do new search.
if (defined($cgi->param('name')))
{
  $module->findTerms(['NamLast', $cgi->param('name')]);
```

```
}
# Otherwise, if id is supplied reattach to existing server-side
# object
elsif (defined($cgi->param('id')))
{
  $module->{id} = $cgi->param('id');
}
# Otherwise, we can't process
else
{
  die("no name or id\n");
}
```

As before, we build a list of columns to fetch:

```
# Build list of columns to fetch
my $columns =
[
  'NamFirst',
  'NamLast'
];
```

If the URL included a `rownum` parameter, fetch records starting from there. Otherwise start from record number 1:

```
# Work out which block of records to fetch
my $rownum = 1;
if (defined($cgi->param('rownum')))
{
  $rownum = $cgi->param('rownum');
}
```

Build the main page as before:

```
# Fetch next five records
my $results = $module->fetch('start', $rownum - 1, 5, $columns);

# Build the results page
print($cgi->header());
print($cgi->start_html());
print($cgi->p("Number of matches: $results->{hits}"));

# Display each match in a separate row in a table
print($cgi->start_table());
foreach my $row (@{$results->{rows}})
{
  print($cgi->start_Tr());
  print($cgi->start_td(), $row->{rownum}, $cgi->end_td());
  print($cgi->start_td(), "$row->{NamFirst} $row->{NamLast}",
    $cgi->end_td());
  print($cgi->end_Tr());
}
print($cgi->end_table(), "\n");
```

Finally we add the **Prev** and **Next** links to allow the user to page backwards and forwards through the results. This is the most complicated part! First, we want to ensure that we connect to the same server and server-side object, so we add the appropriate `port` and `id` parameters to our URL:

```
# Add the Prev and Next links
my $url = $cgi->url();
$url .= '?port=' . $session->{port};
$url .= '&id=' . $module->{id};
```

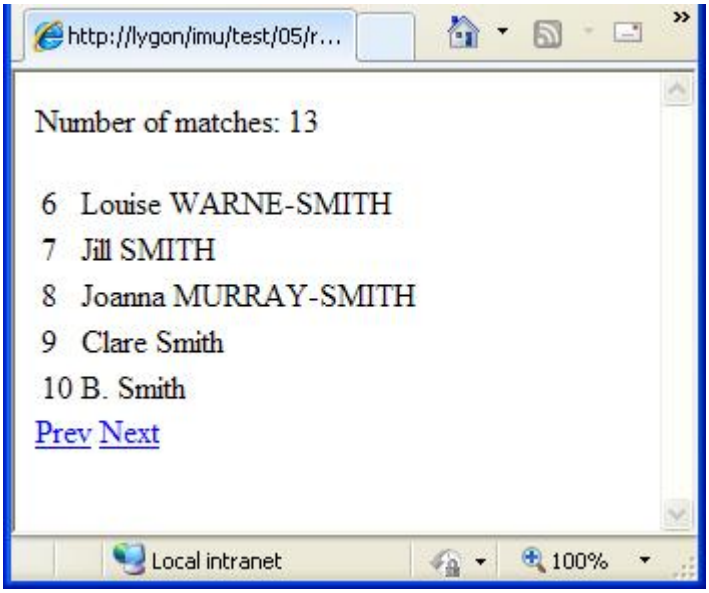
If we are not already showing the first record, we add a **Prev** link to allow the user to go back one page in the result set:

```
my $first = $results->{rows}->[0];
if ($first->{rownum} > 1)
{
   my $prev = $first->{rownum} - 5;
   if ($prev < 1)
   {
      $prev = 1;
   }
   $prev = $url . '&rownum=' . $prev;
   print($cgi->a({href => $prev}, 'Prev'), "\n");
}
```

Similarly, if we are not already showing the last record, we add a **Next** link to allow the user to go forward one page:

```
my $count = @{$results->{rows}};
my $last = $results->{rows}->[$count - 1];
if ($last->{rownum} < $results->{hits})
{
   my $next = $last->{rownum} + 1;
   $next = $url . '&rownum=' . $next;
   print($cgi->a({href => $next}, 'Next'));
}
print($cgi->end_html());
```

The resulting web page looks like this:



SECTION 7

Exceptions

When an error occurs, the IMu Perl API throws an exception (`die` in Perl parlance). The exception is an `IMu::Exception` object.

For simple error handling all that is usually required is to catch the exception and report the exception as a string:

```
eval
{
  …
};
if ($@)
{
  die("Error: $@");
}
```

`IMu::Exception` overrides the Perl "" (stringify) operator (which is called "magically" when the exception object is used as a string) and returns an error message.

To handle specific IMu errors it is necessary to check the exception is an `IMu::Exception` object before using it. `IMu::Exception` includes a property called `id`. This is a string and contains the internal IMu error code for the exception. For example, you may want to catch the exception raised when an `IMu::Session`'s connect method fails and try to connect to an alternative server:

```
my $mainServer = 'server1.com';
my $alternativeServer = 'server2.com';
my $session = IMu::Session->new();
$session->{host} = $mainServer;
eval
{
  $session->connect();
};
if ($@)
{
  if (ref($@) && $@->isa('IMu::Exception'))
  {
    # Check for specific SessionConnect error
    if ($@->getID() eq 'SessionConnect')
    {
      $session->{host} = $alternativeServer;
      eval
      {
        $session->connect();
      };
    }
  }
  # Check for exception again. If we successfully connected
  # to $alternativeServer then $@ will be cleared.
  if ($@)
  {
    die("Error: $@\n");
  }
}
# By the time we get to here the session is connected to either
# the main server or the alternative.
```

SECTION 8

Reference

Module IMu::Handler

```
use IMu::Handler;
```

Provides a general low-level interface to creating server-side objects.

Constructor

```
new([IMu::Session $session])
```

Creates an object which can be used to interact with server-side objects.

Parameters

`$session`	An `IMu::Session` object to be used to communicate with the IMu server. If this parameter is not supplied, a new session is created automatically using the `IMu::Session` class's default host and port values.

Properties

`create`

getter `mixed getCreate()`

setter `setCreate(mixed $create)`

An object to be passed to the server when the server-side object is created. To have any effect this must be set before any object methods are called. This property is usually only set by sub-classes of `IMu::Handler`.

`destroy`

getter `boolean getDestroy()`

setter `setDestroy(boolean $destroy)`

A flag controlling whether the corresponding server-side object should be destroyed when the session is terminated.

`id`

getter `string getID()`

setter `setID(string $id)`

The unique identifier assigned to the server-side object once it has been created.

`language`

getter `string getLanguage()`

setter `setLanguage(string $language)`

The language to be used in the server.

`name`

getter `string getName()`

setter `setName(string $name)`

The name of the server-side object to be created. This must be set before any object methods are called.

`session`

getter `IMu::Session getSession()`

The session object used by the handler to communicate with the IMu server.

Methods

`mixed call(string $method [, mixed $parameters])`

Calls a method on the server-side object.

Parameters

`$method`	The name of the method to be called.
`$parameters`	Any parameters to be passed to the method. The `call` method uses Perl's reflection to determine the structure of the parameters to be transmitted to the server. Passing `$parameters` is optional.

Returns An object containing the result returned by the server-side method.

Throws `IMu::Exception` if a server-side error occurred.

`public mixed request(mixed $request)`

Submits a low-level request to the IMu server. This method is chiefly used by the `call` method above.

Parameters

`$request`	An object containing the request parameters.

Returns A variable containing the server response.

Throws `IMu::Exception` if a server-side error occurred.

Module IMu

```
use IMu;
```

Simple class containing general IMu properties. This class cannot be instantiated.

Class constants

`string` `VERSION`

The version number of the IMu API.

Module IMu::Exception

```
use IMu::Exception;
```

Class for IMu-specific exceptions.

Constructor

```
new(string $id [, mixed $args [, mixed $...]])
```

Creates an IMu specific exception.

Parameters

`$id`	A string exception code.
`$args`	Any additional arguments used to provide further information about the exception.

Properties

`args`

getter `mixed getArgs()`

setter `setArgs(mixed $args)`

The set of arguments associated with the exception.

`id`

getter `string getID()`

The unique identifier assigned to the server-side object once it has been created.

Methods

```
string toString()
```

Used to override the standard Perl "" (stringify) operator.

Returns A string description of the exception.

Module IMu::Module

```
use IMu::Module;
```

Extends: `IMu::Handler`

Provides access to an EMu module.

Constructor

```
new(string $table [, IMu::Session $session])
```

Creates an object which can be used to access the EMu module specified by `$table`.

Parameters

`$table`	Name of the EMu module to be accessed.
`$session`	An `IMu::Session` object to be used to communicate with the IMu server. If this parameter is not supplied, a new session is created automatically using the `IMu::Session` class's default host and port values.

Properties

`table`

getter `string getTable()`

The name of the table associated with the `IMuModule` object.

Methods

`int addFetchSet(string $name, mixed $columns)`

Associates a set of columns with a logical name in the server. The name can be used instead of a column list when retrieving data using `fetch`.

Parameters

`$name`	The logical name to associate with the set of columns.
`$columns`	A string or a reference to an array of strings containing the names of the columns to be used when `$name` is passed to `fetch`. Each string can contain one or more column names, separated by a semi-colon or a comma.

Returns The number of sets (including this one) registered in the server.

Throws `IMu::Exception` if a server-side error occurred.

`int addFetchSets($sets)`

Associates several sets of columns with logical names in the server. This is the equivalent of calling `addFetchSet` for each entry in the map but is more efficient.

Parameters

`$sets`	A hash reference containing mappings between names and sets of columns.

Returns The number of sets (including these ones) registered in the server.

Throws `IMu::Exception` if a server-side error occurred.

`int addSearchAlias(string $name, mixed $columns)`

Associates a set of columns with a logical name in the server. The name can be used when specifying search terms to be passed to `findTerms`. The search becomes the equivalent of an `OR` search involving the columns.

Parameters

`$name`	The logical name to associate with the set of columns.
`$columns`	A string or reference to an array of strings containing the names of the columns to be used when `$name` is passed to `findTerms`. Each string can contain one or more column names, separated by a semi-colon or a comma.

Returns The number of aliases (including this one) registered in the server.

Throws `IMu::Exception` if a server-side error occurred.

`int addSearchAliases($aliases)`

Associates several sets of columns with logical names in the server. This is the equivalent of calling `addSearchAlias` for each entry in the map but is more efficient.

Parameters

`$aliases`	A hash reference containing a set of mappings between a name and a set of columns.

Returns The number of sets (including these ones) registered in the server.

Throws `IMu::Exception` if a server-side error occurred.

`int addSortSet(string $name, mixed $keys)`

Associates a set of sort keys with a logical name in the server. The name can be used instead of a sort key list when sorting the current result set using `sort`.

Parameters

`$name`	The logical name to associate with the set of columns.
`$keys`	A string or a reference to an array of strings containing the names of the keys to be used when `$name` is passed to `sort`. Each string can contain one or more keys, separated by a semi-colon or a comma.

Returns The number of sets (including this one) registered in the server.

Throws `IMu::Exception` if a server-side error occurred.

`int addSortSets($sets)`

Associates several sets of sort keys with logical names in the server. This is the equivalent of calling `addSortSet` for each entry in the map but is more efficient.

Parameters

`$sets`	A hash reference containing a set of mappings between a name and a set of keys.

Returns The number of sets (including these ones) registered in the server.

Throws `IMu::Exception` if a server-side error occurred.

`{}fetch(string $flag, int $offset, int $count [, mixed $columns])`

Fetches `count` records from the position described by a combination of `$flag` and `$offset`.

Parameters

`$flag`	The position to start fetching records from. Must be one of: • `'start'` • `'current'` • `'end'`
`$offset`	The position relative to `$flag` to start fetching from.
`$count`	The number of records to fetch. A `$count` of zero is

permitted to change the location of the current record without returning any results. A `$count` of less than zero causes all the remaining records in the result set to be returned.

`$columns` A string or a reference to an array of strings containing the names of the columns to be returned for each record or the name of a column set which has been registered previously using `addFetchSet`. Each string can contain one or more column names, separated by a semi-colon or a comma.

If this parameter is not supplied, no column data is returned. The results will still include the pseudo-column `rownum` for each fetched record.

Returns A hash reference.

Throws `IMu::Exception` if a server-side error occurred.

```
int findKey(int $key)
```

Searches for a record with the key value `key`.

Parameters

`$key` The key of the record being searched for.

Returns The number of records found. This will be either `1` if the record was found or `0` if not found.

Throws `IMu::Exception` if a server-side error occurred.

```
int findKeys($keys)
```

Searches for records with key values in the array `keys`.

Parameters

`$keys` A reference to the array of keys being searched for.

Returns The number of records found.

Throws `IMu::Exception` if a server-side error occurred.

```
int findTerms([] $terms)
```

Searches for records which match the search terms specified in `$terms`.

Parameters

`$terms` The search terms. The terms are specified using an array reference. Each term is itself an array reference of strings.

Returns An estimate of the number of records found.

Throws `IMu::Exception` if a server-side error occurred.

`int findWhere(string $where)`

Searches for records which match the TexQL `where` clause.

Parameters

`$where` The TexQL `where` clause to use.

Returns An estimate of the number of records found.

Throws `IMu::Exception` if a server-side error occurred.

`int restoreFromFile(string $file)`

Restores a set of records from a file on the server machine which contains a list of keys, one per line.

Parameters

`$file` The file on the server machine containing the keys.

Returns The number of records found.

Throws `IMu::Exception` if a server-side error occurred.

`int restoreFromTemp(string $file)`

Restores a set of records from a temporary file on the server machine which contains a list of keys, one per line. Operates the same way as `restoreFromFile` except that the `$file` parameter is relative to the server's temporary directory.

Parameters

`$file` The file on the server machine containing the keys.

Returns The number of records found.

Throws `IMu::Exception` if a server-side error occurred.

`mixed sort(mixed $keys, mixed $flags)`

Sorts the current result set by the sort keys in `$keys`. Each sort key is a column name optionally preceded by a `"+"` (for an ascending sort) or a `"-"` (for a descending sort).

Parameters

`$keys` A string or reference to an array of strings containing the list of sort keys. Each string can contain one or more keys, separated by a semi-colon or a comma.

`$flags` A string or reference to an array of strings containing a set of flags specifying the behaviour of the sort. Each string can contain one or more flags, separated by a semi-colon or a comma.

Returns An array reference containing the report information if the `report` flag has been specified. Otherwise the result will be undefined.

Throws `IMu::Exception` if a server-side error occurred.

Module IMu::Session

```
use IMu::Session;
```

Manages a connection to an IMu server. The server's host name and port can be specified in the constructor, by setting properties on the object or by setting class-based default properties.

Class Properties

`_defaultHost`

getter `string getDefaultHost()`

setter `setDefaultHost(string $host)`

The name of the host used to create a connection if no object-specific host has been supplied.

`_defaultPort`

getter `int getDefaultPort()`

setter `setDefaultPort(int $port)`

The number of the port used to create a connection if no object-specific host has been supplied.

Constructor

```
new([string $host [, int $port]])
```

Creates a `IMu::Session` object with the specified `$host` and `$port`.

Parameters

`$host` The host to connect to.

If this parameter is not supplied the class property `_defaultHost` will be used instead.

`$port` The port number to connect to.

If this parameter is not supplied, the class property `_defaultPort` will be used instead.

Properties

`close`

getter `boolean getClose()`

setter `setClose(boolean close)`

A flag controlling whether the connection to the server should be closed after the next request. This flag is passed to the server as part of the next request to allow it to clean up. The value `1` (true) indicates that the connection should be closed, `0` (false) that the connection should not be closed.

`context`

getter `string getContext()`

setter `setContext(string context)`

The unique identifier assigned by the server to the current session.

`host`

getter `string getHost()`

setter `setHost(string host)`

The name of the host used to create the connection. Setting this property after the connection has been established has no effect.

`port`

getter `int getPort()`

setter `setPort(int port)`

The number of the port used to create the connection. Setting this property after the connection has been established has no effect.

`suspend`

getter `boolean getSuspend()`

setter `setSuspend(boolean suspend)`

A flag controlling whether the server process handling this session should begin listening on a distinct, process-specific port to ensure a new session connects to the same server process. This is part of IMu's mechanism for maintaining state. If this flag is set to `1` (true), then after the next request is made to the server, the `IMu::Session`'s `port` property will be altered to the process-specific port number.

Methods

`void connect()`

Opens a connection to an IMu server.

Throws `IMu::Exception` if the connection could not be opened.

`void disconnect()`

Closes the connection to the IMu server.

`void login(string $user, string $password [, boolean $spawn])`

Logs in as the given user with the given password. If the `$spawn` parameter is set to `1` (true), this will cause the server to create a new child process specifically to handle the newly logged in user's requests.

Parameters

`$user`	The name of the user to log in as.
`$password`	The user's password for authentication.
`$spawn`	A flag indicating whether the process should create a new child process specifically for handling the newly logged in user's requests. If this parameter is not supplied, it defaults to `1` (true).

Throws `IMu::Exception` if the login request failed.

`{} request({} $request)`

Submits a low-level request to the IMu server.

Parameters

`$request`	A hash reference containing the request parameters.

Returns A hash reference containing the server's response.

Throws `IMu::Exception` if a server-side error occurred.

Index