EMu Documentation

Using KE IMu API (With PHP)

Document Version 1.1

EMu Version 4.0

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 customise IMu components, including:

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

This document provides details of how to create web applications using IMu with PHP.

System Requirements

The IMu PHP library requires PHP version 5.0 or later. Version 5.2 (or later) is recommended as it has many features built-in which must otherwise be added to earlier versions.

The PHP configuration must include the `sockets` and `dom` modules.

If you plan to do server-side document processing using XSLT, the PHP configuration must also include the `xsl` module. If you plan to use AJAX in your website, it is also recommended to include the `json` module.

SECTION 2

Using the IMu PHP library

In order to use the IMu PHP library, include `imu.php` in the PHP code.

For example, if the IMu library is installed in the relative directory:

```
../imu
```

the following line would be added to the PHP code:

```
require_once '../imu/php/lib/imu.php';
```

`imu.php` defines an IMu class. This class includes static members which contain information about the IMu installation. The class includes the following members:

- `IMu::$lib` - the path to the IMu PHP library files.
- `IMu::$lang` - the two letter language code defining the language in which error messages are returned.

The `$lib` member should also be used to simplify other IMu library files.

For example:

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

Exceptions

Many of the methods in IMu library objects throw exceptions when an error occurs. For this reason, all code that uses any IMu library objects should be surrounded with a `try/catch` block.

The following code is a basic template for writing PHP which uses the IMu library:

```
<?php
…
require_once '…/imu.php';
…
try
{
 /* Create and use IMu objects */
 …
}
catch (Exception $error)
{
 /* Handle or report error */
 …
}
```

The exception thrown is an `IMuException` object. `IMuException` is a subclass of the standard PHP `Exception`. In many cases your code can simply catch the standard `Exception` (as in this template). If more information is required about the exact `IMuException` thrown, see *Exceptions* (page 55).

The examples that follow assume that code fragments have been surrounded with code structured in this way.

SECTION 3

Connecting to an IMu server

An `IMuSession` object is used to connect your code to an IMu server.

To use an `IMuSession` object, it is necessary to include IMu's `session.php`.

Several techniques can be used to achieve this:

Pass the hostname and service or port number to the IMuSession constructor

The simplest way to create a connection to an IMu server is to pass the `hostname` and `service` or `port` number to the `IMuSession` constructor and then call the `connect` method.

For example:

```
…
require_once IMu::$lib . '/session.php';
…
$session = new IMuSession('server.com', 12345);
$session->connect();
…
```

Set the public members $host and $port

Alternatively, pass no values to the constructor and then set the public members `$host` and `$port` before calling `connect`:

```
…
require_once IMu::$lib . '/session.php';
…
$session = new IMuSession;
$session->host = 'server.com';
$session->port = 12345;
$session->connect();
…
```

Use the IMuSession class default values

If neither the `host` nor `port` is set, the `IMuSession` class default values will be used. These defaults can be overridden by setting the class variables `$defaultHost` and `$defaultPort`:

```
…
require_once IMu::$lib . '/session.php';
…
IMuSession::$defaultHost = 'server.com';
IMuSession::$defaultPort = 12345;
$session = new IMuSession;
$session->connect();
…
```

This technique is useful when planning to create several connections to the same server or when wanting to get a handler object (page 7) 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 the `IMuHandler` class.

You do not typically create an `IMuHandler` object directly but instead use a subclass.

In this document we examine two of the most frequently used handlers, `IMuModule` and `IMuModules`:

- `IMuModule` allows you to find and retrieve records from a single EMu module.
- `IMuModules` allows you to find and retrieve records from two or more EMu modules.

Section 4

Accessing an EMu Module

An `IMuModule` object must be created in the PHP code in order to access an EMu module. The simplest way to do this is to pass the name of the module to the `IMuModule` constructor.

For example:

```
…
require_once IMu::$lib . '/module.php';
…
$module = new IMuModule('eparties', $session);
```

This code assumes that an `IMuSession` object called `$session` has already been created. If an `IMuSession` object is not passed to the `IMuModule` constructor, a session will be created automatically using the `$defaultHost` and `$defaultPort` class variables. See *Connecting to an IMu Server* (page 5) for details.

Once an `IMuModule` object has been created, it can be used to search the 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:

```
…
require_once IMu::$lib . '/module.php';
…
$module = new IMuModule('eparties', $session);
$hits = $module->findKey(42);
```

The return value is `1` if the record exists and `0` otherwise.

findKeys

The `findKeys` method searches for a set of key values. The keys are passed in a simple PHP array:

```
$module = new IMuModule('eparties', $session);
$keys = array(52, 42, 17);
$hits = $module->findKeys($keys);
```

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.

Each term is specified as a three element array:

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 be searched for.
3. The third element contains a comparison operator to use for the search.

   The operator specifies how the value supplied in the second element of the array should be matched. 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.

findWhere

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

```
…
$module = new IMuModule('eparties', $session);
$where = "NamLast contains 'Smith'";
$hits = $module->findWhere($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:
   `array('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.

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.

Examples

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

```
array('NamLast', 'Smith')
```

Specifying search terms for other types of columns is straightforward. For example, to search for records inserted on `February 13, 2010`:

```
array('AdmDateInserted', 'Feb 13 2010')
```

To search for records inserted before February 13, 2010, it is necessary to add an operator:

```
array('AdmDateInserted', 'Feb 13 2010', '<')
```

To specify more than one search term, create a Boolean `AND` or `OR` term. To do this, create a two element array:

1. The first element of the array is the word `and` or `or`.
2. The second element of the array is an array of other search terms.

For example, to specify a search for Parties records where the first name is `John` and the last name is `Smith`:

```
array('and', array(
 array('NamFirst', 'John'),
 array('NamLast', 'Smith')
))
```

The `and` or `or` terms can be nested. To restrict the previous search to find records inserted before February 13, 2010 or on March 1, 2010, specify:

```
array('and', array(
 array('NamFirst', 'John'),
 array('NamLast', 'Smith'),
 array('or', array(
     array('AdmDateInserted', 'Feb 13 2010', '<'),
     array('AdmDateInserted', 'Mar 1 2010')
 ))
))
```

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

```
…
$module = new IMuModule('eparties', $session);
$term = array('NamLast', 'Smith');
$hits = $module->findTerms($term);
…
```

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

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 Catalogue module for keywords. Our definition of a keywords search is to search the `SummaryData`, `CatSubjects_tab` and `NotNotes` columns. We could do this by simply building an `or` search:

```
$terms = array('or',
array('SummaryData', $keyword),
array('CatSubjects_tab', $keyword),
array('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:

```
$module->addSearchAlias('keywords',
array('SummaryData', 'CatSubjects_tab', 'NotNotes'));
…
$module->findTerms('keywords', $keyword);
```

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. This is useful if the search terms are coming from an HTML form. Suppose we had an HTML form with a keywords text box amongst other search boxes. All we need to do is register the `keywords` alias and then pass the form data directly to the `findTerms` method:

```
$module->addSearchAlias('keywords',
array('SummaryData', 'CatSubjects_tab', 'NotNotes'));
…
$terms = array();
foreach ($_GET as $name => $value)
 $terms[] = array($name, $value);
$module->findTerms(array('and', $terms));
```

To add more than one alias at a time, build an associative array of names and columns and call the `addSearchAliases` method:

```
$alias = array(
'keywords' => array('SummaryData', 'CatSubjects_tab', 'NotNotes'),
'title' => array('SummaryData', 'TitMainTitle'));
$module->addSearchAliases($aliases);
```

Getting Information from Matching Records

IMuModule's fetch method is used to get information from the matching records once a 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 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 an 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 an 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 an array of strings. In its simplest form each string contains a single column name, or several column names separated by commas.

For example, to retrieve the information for both the `NamFirst` and `NamLast` columns, you would pass either:

```
"NamFirst,NamLast"
```

or

```
array("NamFirst", "NamLast")
```

as the `columns` argument to `fetch`. Building the list into a PHP array is convenient if requesting a large number of columns.

Return Value

The fetch method returns records requested in an `IMuModuleFetchResult` object. This object contains two members:

- `hits`
- `rows`

The `hits` member 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` member is an array containing the set of records requested. Each element of the `rows` array is itself an associative array. Each associative array contains members for each column requested. This is probably best demonstrated by PHP's built-in `print_r` function. The following example shows a simple search of the EMu Parties module using `findTerms` with `fetch` used to retrieve a set of records:

```
…
require_once '…/lib/imu.php';
require_once IMu::$lib . '/session.php';
require_once IMu::$lib . '/module.php';
…
try
{
 $session = new IMuSession('server.com', 12345);

 $module = new IMuModule('eparties', $session);

 /* Find all party records where Last Name contains 'smith'
 */
 $hits = $module->findTerms(array('NamLast', 'Smith'));

 /* We want to fetch the irn, NamFirst and NamLast
 ** columns for each record.
 */
 $columns = array();
 $columns[] = 'irn';
 $columns[] = 'NamFirst';
 $columns[] = 'NamLast';

 /* Fetch the first three records (at most) from the start
 ** of the result set.
 */
 $result = $module->fetch('start', 0, 3, $columns);
 print_r($result);
}
catch (Exception $error)
{
 …
}
```

The output of this code will be similar to:

```
IMuModuleFetchResult Object
(
 [hits] => 13
 [rows] => Array
 (
     [0] => Array
     (
         [irn] => 104440
         [NamLast] => SMITH
         [rownum] => 1
         [NamFirst] => Noel
     )
     [1] => Array
     (
         [irn] => 106612
         [NamLast] => Smith
         [rownum] => 2
         [NamFirst] => Alwyn
     )
     [2] => Array
     (
         [irn] => 106985
         [NamLast] => Smith
         [rownum] => 3
         [NamFirst] => Graham
     )
 )
)
```

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 arrays. For example, if a `columns` argument of:

```
"NamLast,NamFirst,NamRoles_tab"
```

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

```
IMuModuleFetchResult Object
(
 [hits] => 1
 [rows] => Array
 (
     [0] => Array
     (
         [NamLast] => Ebb
         [rownum] => 1
         [NamRoles_tab] => Array
         (
             [0] => Lyricist
             [1] => Pianist
         )
         [NamFirst] => Fred
     )
 )
)
```

(Displayed using `print_r`)

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 Catalogue module documents the creator of an object as an attachment (to a record in the Parties module) in a column called `CatCreatorRef`. If the Catalogue module is searched, it is possible to get the creator's last name for each Catalogue 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 associative array:

```
IMuModuleFetchResult Object
(
 [hits] => 1
 [rows] => Array
 (
     [0] => Array
     (
         [irn] => 5
         [CatCreatorRef] => Array
         (
             [NamLast] => Mueck
             [NamTitle] => Mr
             [NamFirst] => Ron
         )
         [rownum] => 1
     )
 )
)
```

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 the set of Catalogue records which attach to the current Parties record via the Catalogue's `CatCreatorRef` column, specify:

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

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

```
…
$module = new IMuModule('eparties', $session);
$hits = $module->findKey(53);

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

$result = $module->fetch('start', 0, 1, $columns);
print_r($result);
```

The output from this fragment illustrates the structure of the `IMuModuleFetchResult` object returned:

```
IMuModuleFetchResult Object
(
 [hits] => 1
 [rows] => Array
 (
     [0] => Array
     (
         [ecatalogue:CatCreatorRef] => Array
         (
             [0] => Array
             (
                 [irn] => 5
                 [TitMainTitle] => In Bed
             )
             [1] => Array
             (
                 [irn] => 50
                 [TitMainTitle] => Man in Blankets
             )
         )
         [irn] => 53
         [NamLast] => Mueck
         [rownum] => 1
         [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 associative array will contain an element called `last_name` rather than `NamLast`.

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:

```
…
$module = new IMuModule('enarratives', $session);

$hits = $module->findKey(2);

$columns = array();
$columns[] = 'irn';
$columns[] = 'NarTitle';
$columns[] = 'NarContributorRef_tab.SummaryData';
$columns[] = 'NarContributorRole_tab';

$result = $module->fetch('start', 0, 1, $columns);
print_r($result);
```

This produces the output:

```
IMuModuleFetchResult Object
(
 [hits] => 1
 [rows] => Array
 (
     [0] => Array
     (
         [NarTitle] => Portrait of William Wilberforce
         [irn] => 2
         [rownum] => 1
         [NarContributorRole_tab] => Array
         (
             [0] => Artist
             [1] => Author
         )
         [NarContributorRef_tab] => Array
         (
             [0] => Array
             (
                 [SummaryData] => Rising, John
             )
             [1] => Array
             (
                 [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 the output of the previous code fragment looks like this:

```
IMuModuleFetchResult Object
(
 [hits] => 1
 [rows] => Array
 (
     [0] => Array
     (
         [NarTitle] => Portrait of William Wilberforce
         [irn] => 2
         [rownum] => 1
         [group1] => Array
         (
             [0] => Array
             (
                 [NarContributorRole_tab] => Artist
                 [NarContributorRef_tab] => Array
                 (
                     [SummaryData] => Rising, John
                 )
             )
             [1] => Array
             (
                 [NarContributorRole_tab] => Author
                 [NarContributorRef_tab] => Array
                 (
                     [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.

`KEModule`’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:

```
$columns = array();
$columns[] = 'irn';
$columns[] = 'NamFirst';
$columns[] = 'NamLast';
$module->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:

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

More than one set can be registered at once using `addFetchSets`. Simply build an associative array containing each set:

```
$sets = array(
 'PersonDetails' => array('irn', 'NamFirst', 'NamLast'),
 'OrganisationDetails' => array('irn', 'NamOrganisation'));
$module->addFetchSets($sets);
```

This technique is very useful when maintaining state (page 33).

A Simple Example

In this example we build a simple PHP-based web page to search the Parties module by last name and display the full set of results.

First build the search page, `search.html`, which is a plain HTML form:

```
<head>
 <title>Party Search</title>
</head>
<body>
 <form action="results.php">
     <p>Enter a last name to search for:</p>
     <input type="text" name="name"/>
     <input type="submit" value="Search"/>
 </form>
</body>
```

Next build the results page, `results.php`, which runs the search and displays the results:

```
<?php
require_once '…/imu.php';

require_once IMu::$lib . '/session.php';
require_once IMu::$lib . '/module.php';

try
{
 $session = new IMuSession('localhost', 45678);
 $module = new IMuModule('eparties', $session);

 /* Build search term and run search.
 ** Search term is passed from search.html using GET
 */
 $text = $_GET['name'];
 $term = array('NamLast', $text);
 $hits = $module->findTerms($term);

 /* Build list of columns to fetch */
 $columns = array
 (
     'NamFirst',
     'NamLast'
 );

 /* Fetch all the matches in one go by passing count < 0 */
 $results = $module->fetch('start', 0, -1, $columns);

 /* Build the results page */
?>
<body>
<p>Number of matches: <?php echo $results->hits ?></p>
<table>
<?php
 /* Display each match in a separate row in a table */
 foreach ($results->rows as $row)
 {
?>
 <tr>
     <td><?php echo $row['rownum'] ?></td>
     <td><?php echo $row['NamFirst'], ' ', $row['NamLast'] ?></td>
 </tr>
<?php
 }
?>
</table>
</body>
<?php
}
catch(Exception $err)
{
 print("Sorry, an error occurred: $err\n");
}
?>
```

The page generated looks like this:

```
Number of matches: 13

1   Noel SMITH

2   Alwyn Smith

3   Graham Smith

4   Peter Smith

5   Kate ECCLES-SMITH

6   Louise WARNE-SMITH

7   Jill SMITH

8   Joanna MURRAY-SMITH

9   Clare Smith

10  B. Smith

11  Ian SMITH

12  Kate Eccles-Smith

13  Grace Cossington SMITH
```

Sorting

The matching set of results can be sorted using `IMuModule`'s `sort` method. This method takes two arguments:

- `columns`
- `flags`

columns

The columns argument is used to specify the columns to sort the result set by. The argument can be either a simple string or an array of strings. Each string can be a simple column name or a set of column names, separated by commas. Each column name can be preceded by a `+` or `–` sign. 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):

```
"+NamPartyType,-NamLast,+NamFirst"
```

-OR-

```
array("+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 `columns` argument, the `flags` argument can be a simple string or an array of strings. Each string can be a single flag or a set of flags separated by 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 behaviour:

- `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 records 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 returned as an array by the `sort` method. The array is hierarchically structured, summarising the number of records which match each of the `sort` keys. See the example below for an illustration of the array 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 array.
  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.

Example

In this example we sort a set of Parties records first by Party Type, then Last Name (descending) and then by First Name:

```
$module = new IMuModule;
$module->findTerms(…);
…
$columns = array[];
$columns[] = '+NamPartyType';
$columns[] = '-NamLast';
$columns[] = '+NamFirst';
$flags = array();
$flags[] = 'full-text';
$flags[] = 'case-sensitive';
$flags[] = 'report';
$summary = $module->sort($columns, $flags);
print_r($summary);
```

This will produce output similar to the following:

```
Array
(
 …
[3] => Array
(
     [count] => 2086
     [value] => Person
     [list] => Array
     (
         …
         [11] => Array
         (
             [count] => 4
             [value] => Young
             [list] => Array
             (
                 [0] => Array
                 (
                     [count] => 1
                     [value] => Derek
                 )
                 [1] => Array
                 (
                     [count] => 1
                     [value] => Don
                 )
                 [2] => Array
                 (
                     [count] => 1
                     [value] => George
                 )
                 [3] => Array
                 (
                     [count] => 1
                     [value] => Shirley
                 )
             )
         )
         …
     )
     …
)
```

From this example we can see that the summary array contains an element for each distinct value in the first sort column (in this case *NamPartyType*). Each element is itself an associative array. The associative array includes a `value` element which contains the distinct value (in this case the value is `Person`) and a `count` element containing the number of the records in the result set which contain this value. The associative array also includes a `list` element. This element contains a summary array of the distinct values for the second sort column (in this case `NamLast`).

SECTION 5

Maintaining State

One of the biggest drawbacks of the earlier example (page 25) is that it fetches the full set of results at one time, which is impractical for large result sets. It is more realistic to display a page of results and allow the user to move forward or backward through the pages.

As any web programmer will be aware however, this simple change of design introduces a significantly higher level of complexity to the implementation, primarily because web pages are *stateless*. The stateless nature of each page leads to many complexities. One of the most common 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 `IMuModule` object to the same server-side object:

```
/* Create a module object as usual */
$first = new IMuModule('eparties', $session);

/* Run a search - this will create a server-side object */
$first->findKeys(array(1, 2, 3, 4, 5, 42));

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

/* Create a second module object */
$second = new IMuModule('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 */
$result2 = $second->fetch('current', 1, 2, 'SummaryData');
```

Although two completely separate `IMuModule` 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 PHP page. 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 `IMuModule` object (to search the module and deliver the first set of results) in one PHP page and then re-connecting to the same server-side object (to fetch a second set of results) in a different PHP page. 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.

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 `false` before calling any of its methods. In the example above, the server would be instructed not to destroy the object as follows:

```
$module = new IMuModule('eparties', $session);
$module->destroy = false;
$module->findKeys(array(1, 2, 3, 4, 5, 42));
```

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 PHP page is executed it connects, effectively at random, to the first process. This process responds to a 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 the 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 page closes the connection to its server, it must notify the server that subsequent pages need to connect explicitly to that process. This is achieved by setting the `IMuSession` object's `suspend` property to `true` prior to submitting any request to the server:

```
$session = new IMuSession('server.com', 12345);
$module = new IMuModule('eparties', $session);
…
$session->suspend = true;
$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 `IMuSession` object's `port` property holds the port number to reconnect to:

```
$session->connection = 'suspend';
$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 previous section 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.php` and is all inside the `try` block (so we don't show the other code outside the `try` block).

First, we create the `IMuSession` object. We set the `port` property to a standard value unless a `port` parameter has been passed in the URL:

```
/* Create new session object.
*/
$session = new IMuSession;
$session->host = 'server.com';

/* Work out what port to connect to
*/
$port = 12345;
if (array_key_exists('port', $_GET))
 $port = $_GET['port'];
$session->port = $port;
```

Next we connect to the server. We immediately set the `suspend` property to `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->suspend = true;
```

We then create the client-side `IMuModule` object and set its `destroy` property to `false`, ensuring the server will not destroy it:

```
/* Create module object and tell the server
** not to destroy it.
*/
$module = new IMuModule('eparties', $session);
$module->destroy = false;
```

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. The
** search term is passed from search.html using GET
*/
if (array_key_exists('name', $_GET))
 $module->findTerms(array('NamLast', $_GET['name']));

/* Otherwise, if id is supplied reattach to
** existing server-side object
*/
else if (array_key_exists('id', $_GET))
 $module->id = $_GET['id'];

/* Otherwise, we can't process */
else
 throw new Exception('no name or id');
```

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

```
/* Build list of columns to fetch */
$columns = array
(
 '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 */
$rownum = 1;
if (array_key_exists('rownum', $_GET))
 $rownum = $_GET['rownum'];
```

Build the main page as before.

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

/* Build the results page */
?>
<body>
<p>Number of matches: <?php echo $results->hits ?></p>
<table>
<?php
/* Display each match in a separate row in a table */
foreach ($results->rows as $row)
{
?>
 <tr>
     <td><?php echo $row['rownum'] ?></td>
     <td><?php echo $row['NamFirst'], ' ', $row['NamLast'] ?></td>
 </tr>
<?php
}
?>
</table>
```

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:

```
<?php
/* Add the Prev and Next links */
$url = $_SERVER['PHP_SELF'];
$url .= '?port=' . $session->port;
$url .= '&id=' . $module->id;
```

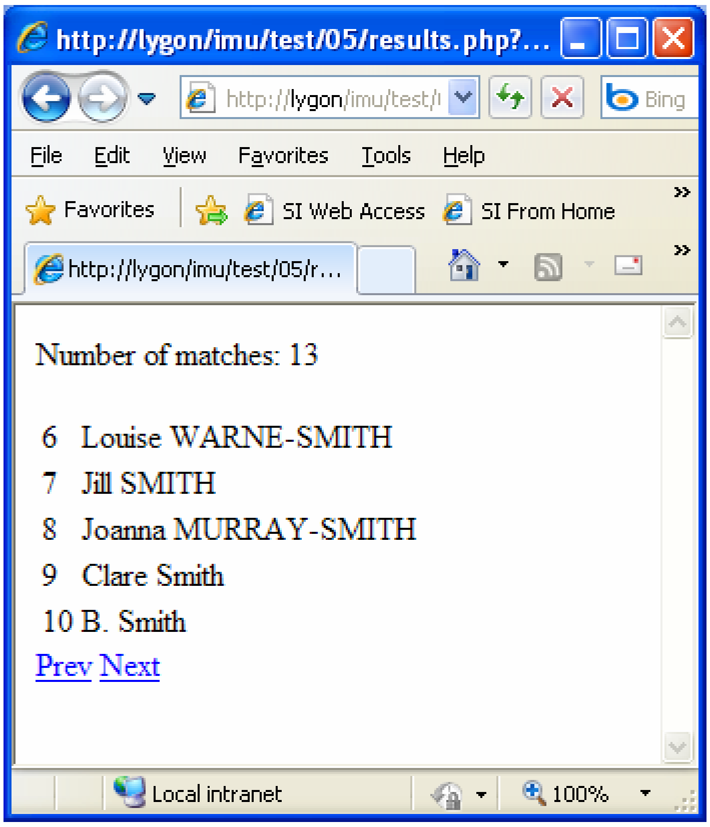
If we are not already showing the first record, we add a **Prev** link to allow the user to go back one page in the result set:

```
$first = $results->rows[0];
if ($first['rownum'] > 1)
{
 $prev = $first['rownum'] - 5;
 if ($prev < 1)
     $prev = 1;
 $prev = $url . '&rownum=' . $prev;
?>
<a href="<?php echo $prev ?>">Prev</a>
<?php
}
```

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

```
$last = $results->rows[count($results->rows) - 1];
if ($last['rownum'] < $results->hits)
{
 $next = $last['rownum'] + 1;
 $next = $url . '&rownum=' . $next;
?>
<a href="<?php echo $next ?>">Next</a>
<?php
}
?>
</body>
```

The resulting web page looks like this:


http://lygon/imu/test/05/results.php?...
File Edit View Favorites Tools Help
Favorites
SI Web Access
SI From Home
Local intranet
100%

Number of matches: 13

```
6   Louise WARNE-SMITH
7   Jill SMITH
8   Joanna MURRAY-SMITH
9   Clare Smith
10 B. Smith
```

Prev Next

SECTION 6

Generating XML

The examples in the previous sections have concentrated on generating HTML directly from the PHP code. However, it may be desirable to use IMu's PHP library to generate XML instead. This may be as part of a web service or to generate web pages using XSLT.

Once information has been retrieved from the IMu server it is of course possible to generate the XML in many ways. The IMu PHP library includes an `IMuDocument` class which may make this a little easier. The `IMuDocument` class is a subclass of the standard PHP `DOMDocument` class (see http://php.net/manual/en/class.domdocument.php for details about `DOMDocument`). The `IMuDocument` class offers some advantages over direct use of `DOMDocument`.

To use `IMuDocument`, you must include the `document.php` file:

```
require_once IMu::$lib . '/document.php';
…
$doc = new IMuDocument;
```

You can set the document encoding by passing the encoding to the constructor:

```
$doc = new IMuDocument('iso-8859-1');
```

The contents of a PHP variable can be added to the XML document using `IMuDocument`'s `writeElement` method. The variable may be a simple value (string, integer, Boolean, etc.), an object or an array.

For example:

```
class Person
{
 public $first;
 public $last;
}
$person = new Person;
$person->first = 'Johann';
$person->last = 'Bach';

$doc = new IMuDocument;
$doc->writeElement('person', $person);
```

will generate XML structured as follows:

```
<person>
  <first>Johann</first>
  <last>Bach</last>
</person>
```

Be aware that `writeElement` has a few quirks when handling arrays. PHP makes no distinction between conventional arrays (with a simple integer index starting from zero) and associative arrays (with a string index). However, when generating XML you almost certainly want to distinguish between these two. When processing an array, `writeElement` checks the set of index values used in the array. If the indexes are all numeric in a range starting from zero, `writeElement` treats the array as a simple list. Otherwise it treats it as an associative array.

The following examples illustrate the difference.

An array used as an associative array:

```
$colours = array();
$colours['red'] = 'Rouge';
$colours['blue'] = 'Bleu';
$colours['green'] = 'Vert';
$doc->writeElement('colours', $colours);
```

will generate XML as follows:

```
<colours>
 <red>Rouge</red>
 <blue>Bleu</blue>
 <green>Vert</green>
</colours>
```

By contrast, an array used as a simple list:

```
$fruits = array();
$fruits[] = 'Apple';
$fruits[] = 'Banana';
$fruits[] = 'Mango';
$doc->writeElement('fruits', $fruits);
```

will generate XML as follows:

```
<fruits>
 <fruit>Apple</fruit>
 <fruit>Banana</fruit>
 <fruit>Mango</fruit>
</fruits>
```

The key difference is that the list is created with repeated sub-elements. The name of the sub-elements is guessed from the name of the main element. In this case the sub-element name is simply the main element name stripped of its trailing `s`. If the name of the sub-element cannot be guessed, `writeElement` uses the name `item`. This behaviour can be overridden by using the `setTagOption` method.

For example:

```
$list[] = 'alpha';
$list[] = 'beta';
$list[] = 'gamma';
$doc->setTagOption('greek', 'child', 'letter');
$doc->writeElement('greek', $list);
```

generates:

```
<greek>
 <letter>alpha</letter>
 <letter>beta</letter>
 <letter>gamma</letter>
</greek>
```

One of the advantages of `writeElement` is that it can be passed an `IMuModuleFetchResult` object returned by `IMuModule`'s `fetch` method.

For example:

```
…
$module = new IMuModule('eparties', $session);
$hits = $module->findKey(53);

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

$result = $module->fetch('start', 0, 1, $columns);
$doc = new IMuDocument;
$doc->writeElement('result', $result);
```

generates:

```
 <result>
 <hits>1</hits>
 <rows>
     <row>
         <ecatalogue:CatCreatorRef>
             <item>
                 <irn>5</irn>
                 <TitMainTitle>In Bed</TitMainTitle>
             </item>
             <item>
                 <irn>50</irn>
                 <TitMainTitle>Man in Blankets</TitMainTitle>
             </item>
         </ecatalogue:CatCreatorRef>
         <irn>53</irn>
         <NamLast>Mueck</NamLast>
         <rownum>1</rownum>
         <NamFirst>Ron</NamFirst>
     </row>
 </rows>
</result>
```

Of course, the names of nodes can be changed by renaming the columns, as described in *Getting Information from Matching Records* (page 15).

Section 7

Searching Several Modules

With IMuModule it is possible to search for and retrieve records from a single EMu module.

It is also possible to have IMu search for information in more than one module and treat the results as a single result set. For example, suppose we want to have a simple search page where the user can enter one or more keywords and search the Narratives, Catalogue and Parties modules, displaying the results as a single result set. The IMu PHP library provides an IMuModules class for this purpose:

```
require_once IMu::$lib . '/modules.php';
…
$modules = new IMuModules($session);
```

Using IMuModules is quite similar to using IMuModule but it requires a little more preparation. First it is necessary to tell IMuModules which modules it should use when searching and retrieving information. This is done with the setModules method.

For example, suppose we want to use the Narratives, Catalogue and Parties modules:

```
$modules->setModules(array('enarratives', 'ecatalogue',
'eparties'));
```

The list of modules passed to setModules also determines the order in which the modules will be searched and the order in which the results will be returned.

Like IMuModule, IMuModules provides methods to search for records:

- findKeys
- findTerms

findKeys

The findKeys method is similar to that in IMuModule. However, the list of keys must include the name of the module as well as its key value:

```
$modules = new IMuModules($session);
…
$keys = array();
$keys[] = array('enarratives', 13);
$keys[] = array('ecatalogue', 42);
$modules->findKeys($keys);
```

findTerms

The `findTerms` method is similar to that in `IMuModule`:

```
$modules = new IMuModules($session);
…
$terms = array('SummaryData', 'loan');
$modules->findTerms($terms);
```

This works the same way as `IMuModule`'s `findTerms` method. However, it is important to realise that the column specified must exist in all the modules being used. If this is not the case, `findTerms` will throw an exception.

For this reason it is often very useful to use search aliases. Suppose a user wants to search for the keyword `design`. You decide that a keyword search means to search the *NarTitle* and *NarNarrative* columns in the Narratives module, the *CreSubjectClassification_tab* and *SummaryData* columns in the Catalogue module and the *BioCommencementNotes_tab* and *SummaryData* columns in the Parties module. This is set up using `addSearchAlias`, in a similar way to `IMuModule`. However, the second argument passed to `addSearchAlias` is an associative array, with the column names to be used for each module. This would be set up as follows:

```
$modules = new IMuModules($session);
…
$aliases = array(
 'enarratives' => array('NarTitle', 'NarNarrative'),
 'ecatalogue' =>
array('CreSubjectClassification_tab','SummaryData'),
 'eparties' => array('BioCommencmentNotes_tab', 'SummaryData')
);
$modules->addSearchAlias('keywords', $aliases);
…
$terms = array('keywords', 'loan');
$modules->findTerms($terms);
```

Both `findKeys` and `findTerms` accept an optional second argument. This argument is a list of modules to be searched. This allows you to restrict the search to just the modules in the list. If no list is supplied, then all the modules given in the call to `setModules` are searched. Be aware that the order of the modules supplied in the list is unimportant. The modules will be searched and the results will be returned in the order of the list given to `setModules`. For example:

```
$modules = new IMuModules($session);
$modules->setModules(array('enarratives', 'ecatalogue',
'eparties'));
…
$aliases = array(
 'enarratives' => array('NarTitle', 'NarNarrative'),
 'ecatalogue' =>
array('CreSubjectClassification_tab','SummaryData'),
 'eparties' => array('BioCommencmentNotes_tab', 'SummaryData')
);
$modules->addSearchAlias('keywords', $aliases);
…
$terms = array('keywords', 'loan');
$modules->findTerms($terms, array('eparties', 'enarratives'));
```

In this example the Catalogue module will not be searched because it is not included in the list passed to `findTerms`. However, the modules will be searched in the order listed in the earlier call to `setModules`, meaning that matching records from the Narratives module will be retrieved before records from the Parties module. Unlike the corresponding methods in `IMuModule`, both `findKeys` and `findTerms` return the list of modules being searched. For the previous example the list would be returned as follows:

```
$list = $modules->findTerms($terms,
array('eparties','enarratives'));
print_r($list);

Array
(
[0] => enarratives
[1] => eparties
)
```

Notice that there is no `findWhere` method.

`IMuModules` provides a `fetch` method similar to the one in `IMuModule`. Setting the columns to be returned by the `fetch` method is not difficult. The idea is to use `addFetchSet` to associate a set of columns with a logical name. This logical name can then be passed to the `fetch` method to retrieve these columns. This is similar to creating a fetch set using `IMuModule`'s `addFetchSet`. There is, however, one important difference. In the case of `IMuModule`, creating a column set is optional and provides a convenient way to refer to a set of columns. When using `IMuModules`, creating fetch sets is the only way to specify which columns should be retrieved for each module.

In this next example we create a column set called `summary`, and for each module we specify which columns are to be returned:

```
$modules = new IMuModules($session);
$modules->setModules('enarratives', 'ecatalogue', 'eparties');
$columns = array
(
 'enarratives' => array('irn', 'NarTitle'),
'ecatalogue' => array('irn', 'SummaryData'),
'eparties' => array('irn', 'SummaryData'),
);
$modules->addFetchSet('summary', $columns);
```

The key point here is that we have mapped three different sets of columns to the same logical name (in this case `summary`).

Once the fetch sets have been added, the `fetch` method can be used to get the results. This works similarly to `IMuModule`'s `fetch`.

For example:

```
$result = $cursor->fetch('start', 0, 5, 'summary');
```

returns the first five records from the result set. The columns returned will be those that match the `summary` column set.

The records in the result set are grouped by module. In the example above, our call to `setModules` added the Narratives module, followed by the Catalogue module, followed by the Parties module. Conceptually this means that the result set comprises all the matching Narratives records followed by all the matching Catalogue records followed by the matching Parties records.

Suppose our `design` search matches three Narratives records, five Catalogue records and four Parties records. This result set effectively looks like this:

Narrative 1
Narrative 2
Narrative 3
Catalogue 1
Catalogue 2
Catalogue 3
Catalogue 4
Catalogue 5
Party 1
Party 2
Party 3
Party 4

This means that when we request the first five records, the result will contain all three matching Narrative records followed by two matching Catalogue records. If we asked for the next page of five records using:

```
$result = $cursor->fetch('current', 1, 5, 'summary');
```

the result will contain the remaining three matching Catalogue records followed by two matching Parties records.

The `fetch` returns an `IMuModulesFetchResult` object. This object contains two members:

- The `count` member contains the total number of records returned.
- The `modules` member contains an array of `IMuCursorFetchModule` objects, one for each module for which records have been retrieved by the `fetch` call. Each `IMuCursorFetchModule` object contains the following members:
  - `name`
  - `index`
  - `hits`
  - `rows`

The `name` member contains the name of the module (such as `eparties`).

The `index` member contains the index of the module in the list of modules registered with using `setModules`. In the example above the index for `enarratives` would be `0`, the index for `ecatalogue` would be `1` and the index for `eparties` would be `2`.

The `hits` member contains the estimated number of matches for the module. This is the same as the `hits` member in `IMuModule`'s `fetch` result.

The `rows` member is an array containing the set of records returned for that module. This is the same as the `rows` member in `IMuModule`'s `fetch` result and is explained in detail in *Getting Information from Matching Records* (page 15).

Example

To illustrate, here is an example `print_r` of a `fetch` result:

```
IMuModulesFetchResult Object
(
 [count] => 5
 [modules] => Array
 (
     [0] => IMuModulesFetchModule Object
     (
         [name] => enarratives
         [index] => 0
         [hits] => 3
         [rows] => Array
         (
             [0] => Array
             (
                 [NarTitle] => James Joule's paddlewheel …
                 [irn] => 1000217
                 [rownum] => 1
             )
             [1] => Array
             (
                 [NarTitle] => Polyhedral sundial
                 [irn] => 120002
                 [rownum] => 2
             )
             …
         )
     )
     [1] => IMuModulesFetchModule Object
     (
         [hits] => 5
         [index] => 1
         [name] => eCatalogue
         [rows] => Array
         (
             [0] => Array
             (
                 [SummaryData] => The designer's handbook
                 [irn] => 2242,
                 [rownum] => 1
             )
             …
         )
     )
 )
)
```

Notice that the result only includes the modules for which the `fetch` retrieved records.

The result set effectively consists of all the records of the first module, followed by all the records of the second module and so on. To access the records for a particular module, pass the name of the module as the `flag` argument to `fetch`.

For example:

```
$cursor->fetch('ecatalogue', 0, 5, 'summary');
```

will `fetch` five records starting from the first `ecatalogue` record. The module names are effectively bookmarks within the result set. Similarly, the module index can be passed as the flag. Remember that the index is the order in which the modules were listed in the call to `setModules`. This means that the following is the equivalent of our previous example:

```
$cursor->fetch(1, 0, 5, 'summary');
```

The `IMuModulesFetchResult` returned by fetch contains three other members: `current`, `prev` and `next`:

- The `current` member contains information about the current record (i.e. the last one returned by the `fetch`).
- The `prev` member contains information about the record that comes immediately before the first record returned by the `fetch`.
- The `next` member contains information about the record that comes immediately after the last record returned.

Each of these members defines a position within the result set. The position is represented by an `IMuModulesFetchPosition` object with two members:

- The `flag` member contains the module name.
- The `offset` member is the offset of the row within that module (starting from `0`).

For example, if we fetch a set of records with:

```
$firstPage = $cursor->fetch(1, 0, 5, 'summary');
```

we can then fetch the next page with:

```
$next = $firstPage->next;
$nextPage = $cursor->fetch($next->flag, $next->offset, 5,
'summary');
```

If there are no records before the first record returned (i.e. we have asked `fetch` to return records from the start of the result set), the `prev` member is not set in the `IMuModulesFetchResult`. Similarly if there are no records after the last record returned (i.e. our request to `fetch` has included the last record in the result set), the next member is not set. This can be used to decide whether to enable **Prev** or **Next** links on a page.

It is useful to understand how `IMuModules` manages the searching of each module. When the modules are registered with the server, the server does not search all of the registered modules immediately. Instead, when `fetch` is called, the server runs just the searches that it needs to satisfy the `fetch` requirements. In our previous example the `fetch` call asked for the first five records. To satisfy this request, the server searches the first module (Narratives) and finds three matches. This is not enough to satisfy the request so the server then searches the next module (Catalogue). It adds the first two records found to the results and returns them. If we then ask for the next five records, the server starts returning records from the third Catalogue record (it does not re-search the Catalogue). There are only three remaining Catalogue record so the server then searches the Parties module and returns the first two matches. The key point here is that the modules are searched only as required. This keeps the server's use of resources to a minimum.

getHits method

`IMuModules` provides a `getHits` method which is useful for identifying how many matches there may be for each module. When the name of the module is passed to this method, the estimated number of matches is returned. If a module name is not passed to `getHits`, the total number of estimated matches (across all modules) is returned:

```
$partyHits = $cursor->getHits('eparties');
$totalHits = $cursor->getHits();
```

If you pass the name of a module that was not listed in the call to `setModules`, `getHits` throws an exception. If you pass the name of a module that has been excluded from the current search (by not including it in the list passed as the second argument to `findKeys` or `findTerms` as described above), `getHits` will return a value of `-1`.

S E C T I O N 8

Exceptions

When an error occurs, the IMu PHP library throws an exception. The exception is an `IMuException` object. This is a subclass of PHP's standard `Exception` class.

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

```
try
{
 …
}
catch (Exception $e)
{
 echo "Error: $e";
 exit(1);
}
```

`IMuException` overrides the `Exception's` `__toString` method (which is called "magically" when the exception object is used as a string) and returns an error message. The message returned is in the language defined in `IMu::$lang`.

Ideally `IMuException` would override `Exception`'s `getMessage` method to return the error message. Unfortunately, `getMessage` is declared `final` in `Exception`, preventing it from being overridden.

To handle specific IMu errors it is necessary to catch the exception as an `IMuException` object. `IMuException` includes a public member 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 `IMuSession`'s connect method fails and try to connect to an alternative server:

```
$mainServer = 'server1.com';
$alternativeServer = 'server2.com';
$session = new IMuSession;
$session->host = $mainServer;
try
{
 $session->connect();
}
catch (IMuException $e)
{
 /* Check for specific SessionConnect error
 */
 if ($e->id != 'SessionConnect')
 {
    echo "Error: $e";
    exit(1);
 }
 $session->host = $alternativeServer;
 try
 {
    $session->connect();
 }
 catch (Exception $e)
 {
    echo "Error: $e";
    exit(1);
 }
}
/* By the time we get to here the session is connected
** to either the main server or the alternative.
*/
```

`IMuException` includes a `getString` method. This method takes a `$lang` parameter and returns an error message in the requested language. If a `$lang` value is not passed, the message returned is in the language defined in `IMu::$lang`.

The error messages are defined in the file `strings.xml` which is in the shared directory of the IMu installation.

Index