IMu Documentation

Developing Web Pages with IMu Web

Document Version 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.

The core of IMu is a back-end server and a set of APIs which provide access to EMu data and multimedia.

IMu Web is functionality (built using the PHP API) to provide a standard but configurable (*out-of-the-box*) set of web pages for browsing and searching EMu collections.

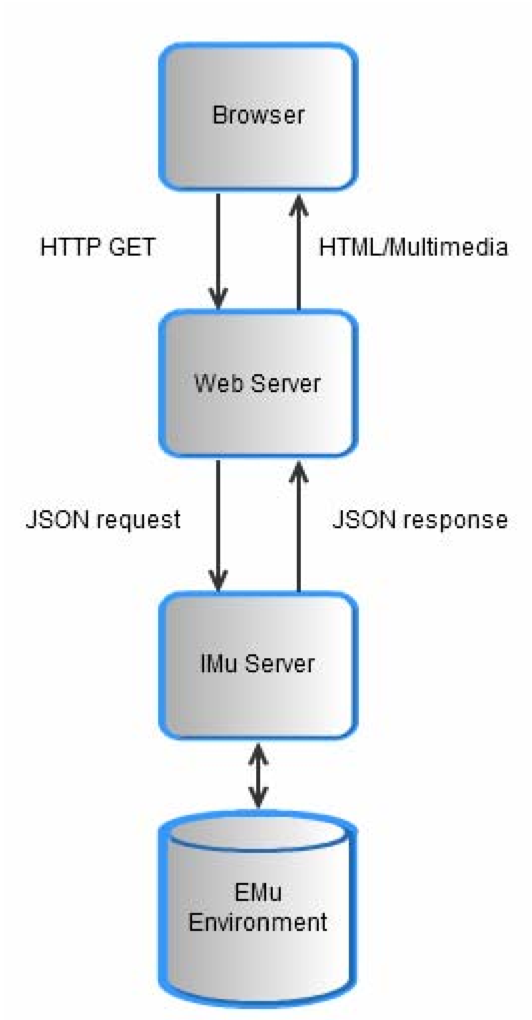
This document provides details of how IMu Web works, how it can be configured and how to deploy IMu Web to client sites.

SECTION 2

Overview

IMu Web's pages use a conventional three-tier web-based architecture:

- A browser requests a page from a Web Server running IMu Web.
- IMu Web in turn requests information from an IMu Server running in an EMu environment.
- This information is then marked up as HTML and returned to the browser.



The Web Server and the IMu Server may run on separate machines or on the same machine. If they run on separate machines, which is common when the Web Server machine sits in a *Demilitarised Zone* (DMZ), ensure that any firewall between the Web Server machine and IMu Server machine does not prevent proper operation. This is explained in more detail below.

Processing a request

IMu Web’s processing of a request is quite complex. The following provides an outline of the steps typically involved in accepting the request and returning the results to the browser:

1. The browser submits its request using a conventional `HTTP GET`. All IMu Web pages are requested using URLs with the form:

   `http://`*web-server*`/`*path*`/imu.php?request=`*type...*

   All requests invoke the same initial piece of code, `imu.php`. The first parameter passed to this code is the *type* of the request. Any additional parameters are passed after the request *type*.
2. The web server invokes its PHP module to execute `imu.php`.

IMu works with standard web servers which can run PHP. This includes Apache and Microsoft’s Internet Information Server (IIS). No specialised web engine is required.

3. The `imu.php` code checks the request *type* and loads a request-specific file. This file defines a class and uses it to create an object to handle the particular type of request. Usually the request object uses IMu's PHP API to submit the request to the appropriate IMu server (there can be multiple IMu Servers running on the one machine, each for a different web environment).
4. The IMu server processes the request and returns data to the requesting PHP object.
5. The PHP object marks up the returned data as XML.
6. The PHP object locates the appropriate XSLT stylesheet associated with the request and uses PHP’s XSLT processor to apply the stylesheet to the XML document.
7. The resulting HTML generated by the previous step is returned to the browser. The HTML includes references to CSS stylesheets which are used to apply the look and feel to the page. The HTML may also include references to JavaScript files. The browser then runs the JavaScript to apply effects such as page transitions and additional markup.

SECTION 3

Development Setup

To develop IMu Web pages, IMu must be set up on a development machine. Set up includes installation and configuration of:

1. The IMu Server within the appropriate EMu back-end environment
2. The IMu software
3. A Web Server

IMu Server

All IMu applications require an IMu server to be running in the appropriate EMu back-end environment (there can be multiple IMu Servers running on the one machine, each for a different web environment). The IMu server (called `imuserver`) is run within an EMu back-end environment as a standard EMu load.

This document describes the development of pages for a back-end environment running EMu version 4.0.02 or later. This version of EMu includes an appropriate version of the IMu server.

The role of `imuserver` is to listen for connections from IMu API programs, accept requests and respond. The server runs as either a single process or can start several child processes.

Before the IMu server can be started it must be configured. IMu server is configured by editing the `etc/imuserver.conf` file in the EMu environment. Each environment will contain an `etc/imuserver.conf.sample` file which can be copied and edited.

The following configuration parameters will almost certainly need to be set:

`main-port`	This is the port number that `imuserver` will use to listen for initial connections from the IMu API. By convention, on a development machine this number is set to `20000` more than the client's standard EMu port number. For example, the Art Demo client uses a port number of `20136`. This means that the value of `main-port` for Art Demo `imuserver` would usually be `40136`. If this value is not set, it will default to `40000`.

`data-source`	This parameter is used by `imuserver` to determine how to connect to a texserver running in the appropriate EMu back-end environment. The value can be: • A port number or a service name • A host name and a port number/service name (as `host:port`) • An empty string If this value is not set, it will default to a port number `20000` less than the value used for `main-port`. If the value contains a port number or service name, `imuerver` uses it to create a socket connection to a texserver process, which is the same way that the EMu client creates its connection. If a host name is not included with port number or service name, the connection will be made to the local machine (`127.0.0.1`). If the value is empty (i.e. set to an empty string), the IMu Server will use a pipe to connect to texserver, rather than a socket. If a socket connection is used, `imuserver` will try to log in as the user specified in the `user-name` entry (see below). If no `user-name` is specified, `imuserver` will try to log in as the user running the `imuserver` (almost certainly user `emu`). It is very important to be aware that for security reasons `imuserver` does not store the user's password. This means that it must be possible to authenticate the user passed by `imuserver` to texserver without the use of a password. This is achieved by enabling `rhosts` authentication on the server machine.

The following settings are less likely to require modification:

`user-name`	This is the name of a user that `imuserver` will pass to texserver for authentication (see `data-source`). This only applies if `imuserver` is using a socket connection to texserver. Be aware that this user must be able to be authenticated by texserver without requiring a password.

`visibility`

This value is used to control which records `imuserver` can retrieve when searching EMu tables. The value can be:

- `internet`

  Only match records with the `AdmPublishWebNoPassword` column set to `Yes`.
- `intranet`

  Only match records with the `AdmPublishWebPassword` column set to `Yes`.
- `all`

  Match any records.

If this value is not set, it will default to `internet`.

`process-count`

This is the number of processes that `imuserver` will create to listen for connections. In environments where there may be a high number of requests it is useful to have several processes listening for connections and handling requests concurrently.

> Each process that is started connects to a texserver. This means that each process will use an EMu licence slot.

`reconnect-port`

This is the port number that each server process will start at when trying to allocate a unique port for handling client reconnections.

For example, suppose that `process-count` is set to `3` and that `reconnect-port` is set to `50000`. When imuserver starts, three processes will be created. Each process will independently attempt to allocate a unique port to listen for reconnections. All three processes will first attempt to allocate port 50000. One will succeed and the other two will fail. The two unsuccessful processes will then move on to try to allocate port 50001. One will succeed and the other will fail. The unsuccessful process will then allocate port 50002.

Reconnection to a server is explained in detail in the Maintaining State section of *Using KE IMu API (With PHP)*.

If this value is not set, it defaults to `5000` more than the value of `main-port`.

`admin-port`	This is the number of a secondary port that `imuserver` will also listen on for connections. The difference between this setting and `main-port` becomes significant when `imuserver` is running several child processes to handle requests. A connection on `main-port` may be handled by any of the processes (which is typically what you want), whereas a connection on `admin-port` is guaranteed to be accepted only by the master process which controls all the other processes. This makes this useful for administering the `imuserver` itself, hence the name `admin-port`. If this value is not set, it defaults to `10000` more than the value of `main-port`.
`language`	This is the two-letter code for the language to be used if none is supplied by the API. If this value is not set, it defaults to `en`.
`handler-timeout`	When IMu web (and other IMu client applications) use functionality in the IMu server, it causes the server to allocate server-side objects which are known as handlers. These handlers are maintained by the server until the client advises the server to destroy them. However, with stateless applications such as IMu Web the client API may never get a chance to advise the server to destroy the handlers. This can result in handlers being maintained by the server when they are not really required. This can impose a significant memory overhead on the server machine. For this reason the server will automatically destroy handlers which have not been used for a certain time. The `handler-timeout` parameter specifies how long a handler can remain unused for before it is automatically discarded by the server. The value is specified as a number followed by a letter `h` (for hours), `d` (for days) or `w` (for weeks). Handlers are described in detail in *Understanding the KE IMu Server*. If this value is not set, it defaults to `1h` (one hour).

`context-timeout`	Each time IMu Web (and other IMu client applications) create a new connection to the IMu server, the server process allocates a new connection "context". The context maintains information about the connection, including the texserver which it is using and a list of all the active handlers. The context is usually destroyed (including all its handlers) and the associated texserver is shutdown when the client application terminates the connection. However, with stateless applications such as IMu Web the client API may never get a chance to terminate the connection. This can result in connection contexts being maintained by the server when they are not really required. For this reason the server will automatically destroy connection contexts which have not been used for a certain time. The `context-timeout` parameter specifies how long a connection context can remain unused for before it is automatically discarded by the server. The value is specified as a number followed by a letter `h` (for hours), `d` (for days) or `w` (for weeks). If this value is not set, it defaults to `3h` (three hours).
`temp-path`	This is the path to a directory where `imuserver` creates temporary files. If the value is not set, it defaults to the same location directory as used by texserver for its temporary files.
`trace-file`	The is the name of a file used to record server-side tracing information. A value of `STDOUT` causes all information to be written to `imuserver`'s standard output. This is normally what is wanted if `imuserver` is started as an `emuload`. If the value is not set, it defaults to `STDOUT`.
`trace-level`	This number indicates the amount of server-side tracing information which is generated. A higher number will generate more information. A value of `1` will generate minimal output and a value of `0` will cause no information to be generated at all. If the value is not set, it defaults to `1`.
`trace-prefix`	This specifies the format of text added to the start of each line of tracing information.

Once configured, the `imuserver` can be started. The server can be started as follows to test that the configuration is working:

```
emu-artdemo@lygon[1] imuserver
2010-09-15 11:31:24: 19761 imuserver started with the following
configuration:
{
  'trace-file' => 'STDOUT',
  'visibility' => 'internet',
  'language' => 'en',
  'trace-level' => 1,
  'data-source' => 20136,
  'admin-port' => 50136,
  'handler-timeout' => 3600,
  'context-timeout' => 10800,
  'reconnect-port' => 45136,
  'main-port' => 40136,
  'process-count' => 1,
  'temp-path' => '/tmp/texpress',
  'trace-prefix' => '%D %T: %p '
}
```

This output indicates that the server is listening for connections and shows the configuration parameters that have been set.

The server is then ready to be started using the standard `emuload`:

```
emu-hv@lygon[2] emuload start imu
fifo                    alive           19 Aug 2010 15:42:35
imu             19783   started         15 Sep 2010 11:34:33
```

The imu load creates a log file (which is the trace output of `imuserver`) in the `loads/imu/logs` directory. As with other loads the name of the log file is the date and time the load was started. However, the imu load also creates a link to the file called `current`, which makes it easy to look at the current `imuserver` tracing:

```
emu-hv@lygon[3] cd loads/imu/logs
emu-hv@lygon[4] tail -f current
2010-09-15 11:34:32: 19783 imuserver started with the following
configuration:
{
  'trace-file' => 'STDOUT',
  'visibility' => 'all',
  'language' => 'en',
  ...
}
...
```

Installing the IMu software

To use and create IMu Web pages in a development environment an `imu` user account needs to be created on the development machine. The `imu` user account should belong to an `imuadmin` group. This is directly analogous to the `emu` user in group `emuadmin` used for managing EMu back-end environments and doing EMu development.

Typically the `imu` and `emu` user accounts will exist alongside one another on the one development machine. However, there should be no reliance of one upon the other.

Once the `imu` user account has been created the IMu client software can be checked out. First log in as user `imu` and then use CVS.

Log in:

```
$ touch $HOME/.cvspass
$ CVSROOT=:pserver:user@cvs.mel.kesoftware.com:/home/cvs/imu
$ export CVSROOT
$ cvs login
Logging in to:
 pserver:andrew@cvs.mel.kesoftware.com:2401/home/cvs/imu
CVS password:
```

Checkout the IMu environment:

```
$ cd $HOME/..
$ cvs checkout imu
cvs checkout: Updating imu
U imu/.cvsignore
U imu/.profile
cvs checkout: Updating imu/master
U imu/master/.cvsignore
U imu/master/.profile
cvs checkout: Updating imu/master/bin
...
```

Three directories will be created in `imu`'s home directory:

- `CVS`
- `master`
- `tools`

The CVS directory is used by CVS itself. The other two are similar in function to the equivalent directories in the EMu back-end environment.

The `master` directory contains a template for developing clients-specific pages in IMu Web. The following directories are included in the `master` directory:

`bin`

This includes scripts for the deployment of imu (`imubldsetup` and `imuinstall`). These are described in more detail below.

`lib`

> This contains the code for the IMu APIs. The APIs currently distributed for IMu include those for .Net (in the dotnet directory) and PHP (in the php directory). More information on using the APIs is available in *Using KE IMu API (With PHP)*.

`web`

> This contains the code for IMu Web. Development of pages using IMu Web is covered in this document.

`ws`

> This contains the code for web services built using IMu.

Development of IMu Web pages is done in a similar way to the development of EMu back-end environments. A client-specific directory is created alongside the `master` directory and the appropriate files are "sync’d" from the `master` directory to the client-specific directory.

One difference is that unlike EMu environments, the client-specific directories used by IMu are, by default, given the name of the EMu Catalogue which the client uses. For example, the Art Demo client uses a Catalogue called `eart`. To do web page development for Art Demo you should use a directory called `eart`.

Setting up the client-specific directory is similar to the way it is done in EMu. First you create the directory and then run the `imusync` command. If you want to do IMu Web development, you also need to create an empty web directory so `imusync` knows to sync the web components as well.

For example:

```
$ cd ~imu
$ mkdir eart
$ mkdir eart/web
$ imusync -c eart
Updating subpart bin...
    eart...
Updating subpart lib...
    eart...
Updating subpart web...
    eart...
Updating subpart work...
    eart...
```

This will link the appropriate files into the `eart` directory. You can then switch to the `eart` environment:

```
$ client eart
Current client is eart
$ ls
bin  lib  web  work
```

The environment is now set up.

Setting up a Web Server

The final step in setting up the development environment is to have an appropriate web server installed. IMu Web requires a web server which can run PHP version 5.0 or later. This includes standard servers such as Apache and Microsoft's IIS.

IMu Web requires the version of PHP installed to include the following modules:

- `ctype`
  Required for processing the data transmitted between the PHP API and the IMu server.
- `dom`
  Required for processing XML configuration files and for generation of XML data.
- `json`
  Required for generating JSON formatted output to be returned to the browser (usually as part of an AJAX request).
- `xsl`
  Required for processing the XML data and transforming it into HTML using XSLT stylesheets.

Using PHP 5.3 or later is preferred because all of these modules are included by default. Versions 5.0, 5.1 and 5.2 require these modules to be added separately. The correct way to add these modules differs depending upon the operating system and the web server being used.

Once the web server has been installed and PHP enabled and configured to use the required modules, all that remains is to create a mapping between a URL and the base directory containing the externally accessible files used to process IMu Web requests. As mentioned before, all IMu Web pages have a URL of the form:

`http://`*web-server*`/`*path*`/imu.php?request=`*type...*

The `imu.php` file is in the environment's `web` directory. For example, for the `eart` environment the `imu.php` file is in `~imu/eart/web/imu.php`. This means that the web server needs a mapping between the *path* component of the URL and the environment's `web` directory. Suppose we want to serve pages from our `eart` environment with a URL such as:

`http://`***web-server***`/artdemo/imu.php?request=`***type***`...`

and that the environment was installed in

```
/home/ke/imu/eart
```

We need to create a mapping between these.

In Apache this is done by either:

- Creating an alias such as
  ```
  Alias    /artdemo/ "/home/ke/imu/eart/web/"
  ```

-OR-

- Creating a symbolic link from the document root such as
  ```
  ln –s /home/ke/imu/eart/web /path-to-document-root/artdemo
  ```

SECTION 4

How a request is processed

In this section we look in more detail at how IMu Web processes requests. By way of illustration we look at the `eart` system. The correct setup of an IMu environment is described in Development Setup (page 5).

When a browser requests a page from IMu Web the web server runs the code in the environment’s `web` directory’s `imu.php` (e.g. `~imu/eart/web/imu.php`). The code in `imu.php` performs the following steps:

1. Loads the IMu API.
2. Enables tracing so that the processing of requests is logged.
3. Creates a list of request parameters.
4. Loads a separate file corresponding to the `request` parameter.
5. Creates an object of the request-specific class.
6. Call the object’s `configure` method to load configuration settings.
7. Calls the object’s `process` method to do the actual request processing.

Each of these steps is described below.

Loading the IMu API

The IMu API files are in IMu's `lib` directory. The API is documented in *Using KE IMu API (With PHP)*.

Tracing

Tracing or logging (the terms are used interchangeably) information is written to a file called `trace.txt`. The file is in the same directory as `imu.php`. In our `eart` example this corresponds to `~imu/eart/web/trace.txt`.

This file is written to by the web server and must therefore have the appropriate permissions. Whilst IMu's tracing can create the trace file automatically it is very likely (and a good thing) that the web server will not have permission to create files in the environment's `web` directory. For this reason the file should be created manually. Once the file has been created the web server will still need permission to write to it.

For unix systems running a server such as Apache there are two ways to do this:

- Give all users write access to the file

```
$ touch trace.txt
$ chmod 666 trace.txt
$ ls -l trace.txt
-rw-rw-rw- 1 imu imuadmin 0 Sep 22 12:18 trace.txt
```

-OR-

- Put the file in the same group as the web server and give the group write access to the file. For example, if the web server runs in the group `webservd`:

```
$ touch trace.txt
$ chgrp webservd trace.txt
$ chmod 660 trace.txt
$ ls -l trace.txt
-rw-rw-rw- 1 imu imuadmin 0 Sep 22 12:18 trace.txt
```

The advantage of the first method is that it is simple to set up. The disadvantage is that it allows all users on the machine to see and even alter the trace information. The second method avoids this by restricting access to the `imu` user and any user in the same group as the web server (which should be no one). The disadvantage of the second method is that on most systems it requires root access to be able to change the group of the file.

Once tracing is enabled each request is recorded in the trace file. The amount of information recorded can be configured (see below). At a minimum the trace will show:

- The date and time of request.
- The IP address of the requester.
- The specific type of request.
- Any additional request parameters.
- The name of the file and class which will be used to handle the request.

Here is a sample of trace information for a browse request:

```
2010-09-22 12:30:08: BEGIN
2010-09-22 12:30:08: request from 172.16.57.34
2010-09-22 12:30:08: request: browse
2010-09-22 12:30:08: file: common/browse.php
2010-09-22 12:30:08: class: IMuBrowseRequest
2010-09-22 12:30:08: script: /artdemo/imu.php
2010-09-22 12:30:08: base: /artdemo
2010-09-22 12:30:08: params:
2010-09-22 12:30:08: end params
2010-09-22 12:30:08: url: /artdemo/imu.php?request=browse
2010-09-22 12:30:09: END
```

Request parameters

The parameters can be passed using either HTTP's `GET` or `POST`. This allows the page designer to choose which technique can be used for submitting forms.

Loading the request-specific file

Every IMu Web request must include a `request` parameter. This identifies what type of request is being submitted. If a `request` parameter is not included, `imu.php` will generate an error.

The `request` parameter is used to find the file that contains the code used to handle the specific request.

To understand how the file is located it is necessary to understand how the environment's `web` directory is organised. In addition to the two files that we have looked at already (`imu.php` and `trace.txt`) the directory may also contain up to three sub-directories. When IMu Web needs to find a file it looks in these directories:

- `common`

  Contains files that are common to all IMu Web environments. These files are managed by KE. They are present in the IMu master environment and changes to these files are controlled using CVS.
- `client`

  Contains files that are specific to the client environment. In our `eart` example, the `client` directory contains files that are specifically used for building pages that work for the `eart` Catalogue. These files are also managed by KE. They are present in the IMu master environment and changes to these files are controlled using CVS.
- `local`

  Contains files that may be used at a particular customer's site. These files are not managed by KE. They are not present in the master environment and are not controlled by CVS. An upgrade of IMu Web will not alter files in the `local` directory in any way.

When IMu Web needs to locate a particular file it looks first in the `local` directory then the `client` directory then the `common` directory. If the file is still not found, IMu Web will raise an error.

Loading files in this way allows a `client` implementation to override a `common` file and a `local` file to override a `client` file.

IMu Web uses this method when loading the file to handle the request. For example, if a browser submits a browse request:

```
http://server/artdemo/imu.php?request=browse
```

`imu.php` first looks for a file called `local/browse.php`, then `client/browse.php` and then `common/browse.php`. In a conventional IMu Web environment the request handling code is all general purpose and so is contained in the `common` directory. This means that when the browse request is submitted IMu will find a corresponding `common/browse.php` file.

`imu.php` loads the code in this file simply by using PHP's standard require mechanism. The file contains a class definition. This class must be a subclass of the `IMuWebRequest` base class. The file must also set a global variable called `$class` with the name of the class defined in the file. IMu Web uses this variable to know what class of object to create (see below).

Here is an outline of the `browse.php` file:

```
<?php
require_once dirname(__FILE__) . '/request.php';

$class = 'IMuBrowseRequest';
class IMuBrowseRequest extends IMuWebRequest
{
    public function
    process()
    {
      …
    }
}
```

Creating a request-specific object

Once the request-specific file has been loaded it is used to create an object to handle the request.

The request handling class must be a subclass of the `IMuWebRequest` base class. This base class is defined in the `common/request.php` file. It provides useful functionality for dealing with the request, including:

- Loading additional configuration settings.
- Getting and setting additional request parameters.
- Finding the first or all files with a common name in the `local`, `client` and `common` directories.
- Creating a connection to the IMu server.
- Creating DOM document objects.
- Creating XSLT processor objects.

Configuration

After the code in `imu.php` has created the request-handling object it calls the object's `configure` method, which is used to configure the request handler. The base class's `configure` method loads information from a configuration file called `config.xml`.

The `configure` method looks for a `config.xml` file in each of the `common`, `client` and `local` directories in turn. This allows standard configuration information to be specified in the `common/config.xml`, client-specific configuration information to be specified in `client/config.xml` (possibly overriding values set in `common/config.xml`) and site-specific configuration information to be specified in `local/config.xml` (possibly overriding values from the other files).

The format of the `config.xml` file is straightforward. As the name implies it is an XML file formatted as follows:

```
<?xml version="1.0"?>
<config>
    <default-lang>en</default-lang>
    <default-theme>london</default-theme>
    <default-view>list</default-view>
    <server-port>40136</server-port>
    <trace-level>1</trace-level>
    <use-javascript>yes</use-javascript>
</config>
```

The following values can be defined in the `config.xml` file:

`cookie-duration`	IMu Web uses cookies to store information about a user. This information is used to record user preferences and hold the IRN of a record in the `ewebusers` table to re-identify the user. This value controls how long the cookie remains valid. The value is a number followed by a single letter `h` (for hours), `d` (days) or `w` (weeks). If the value is not set, it defaults to `7d`.
`cookie-name`	This is the name of the cookie set in the browser. There is usually no need to change this as the cookie is applied only to a specific domain. If the value is not set, it defaults to `IMu`.
`default-lang`	This is the two-letter code for the language to be used. If the value is not set, it defaults to `en`.
`default-theme`	The name of the theme to be displayed by default. Themes are explained in detail below.

`default-view`	The name of the view to be displayed by default. Views are explained in detail below.
`google-maps-key`	IMu Web can display the location of specimens on Google Maps. To be able to use the Google Maps API a site must submit a key. This setting holds the key value. Google Maps keys are freely available from Google at http://code.google.com/apis/maps/signup.html.
`server-host`	This is the name or IP address of a host machine running `imuserver` that IMu Web will connect to. If the value is not set, it will default to `127.0.0.1`
`server-port`	This is the port number that IMu Web will use to connect to a machine running `imuserver`. If the value is not set, it will default to `40000`.
`trace-level`	This number indicates the amount of client-side tracing information that is generated. A higher number will generate more information. A value of `1` will generate minimal output and a value of `0` will cause no information to be generated at all. If the value is not set, it will default to `1`.
`use-java-script`	This value controls whether the pages returned by IMu Web will run JavaScript. The value must be `yes` or `no`. If the value is not set, it will default to `yes`.
`use-yahoo-maps`	This value controls whether IMu Web's mapping tools will use Yahoo maps. The value must be `yes` or `no`. If the value is not set, it will default to `yes`.

Processing

The final step in handling a request is running the object's `process` method. This method is where the actual request-specific processing takes place. Each of the request classes provided with IMu Web includes a `process` method that is called by `imu.php`.

The `process` method is responsible for interpreting the request, forwarding parts of the request to the IMu server and generating the results. As such, `imu.php` effectively operates as a simple web service. The results of the web service are not generic data marked up as XML however, but data targeted explicitly for display in a browser.

The standard IMu Web `common` directory includes code for handling the following types of request. Each of these requests generates a browser page:

`browse`	Implements the Browse Collections page.
`search`	Implements the Search Collections page.
`display`	Returns a paginated result set for display. If the user is using JavaScript, results from this page are appended to the Search Collections page rather than being displayed in a separate page.
`museum`	Implements the My Museum page.
`collection`	Generates a page for changing the details of a particular collection (part of My Museum).

In addition to these files the `common` directory also includes code for other processing. These requests do not generate a browser page:

`multimedia`	Returns a multimedia resource which has been retrieved from the IMu server. If the resource is an image, it may be resized or reformatted dynamically.
`load`	Generates a result set. The result set may be generated by a search, by restoring a saved set of records or by loading a set of records listed in an attachment column. It is these results that the `display` request fetches for display.
`hits`	Retrieves the number of matches for each module included in a `load` request.
`group`	Adds matches to or removes matches from a user’s collection.
`config`	Manages the user’s configuration. This includes the preferred language and theme and whether or not to use JavaScript with the pages.

Most of these types of request use the IMu Server to run a search, fetch data from an EMu table or retrieve multimedia. This document does not describe in detail how IMu Web interacts with the IMu server - for details about how to use the IMu PHP API see *Using KE IMu API (With PHP).*

Once a request has retrieved information from the IMu Server it must mark it up for return to the browser. Requests such as `browse`, `search` and `display` must ultimately produce HTML that can be displayed in the browser. There are several ways this might be done, and each with advantages and disadvantages:

Direct HTML Generation	Generate HTML directly from the request's `process` method. This is the conventional PHP way to generate output. Direct HTML Generation is simple for a PHP programmer to understand. It is also probably more efficient in terms of page generation speed. However, it does not provide any separation of the data acquisition from the creation of the actual user-interface (the HTML). This makes it difficult to implement different views of the same data without code replication.
Browser-based XSLT Transformation	Mark up the data as XML and return it to the browser along with an embedded reference to an XSLT stylesheet. The browser then fetches the stylesheet and uses it to transform the XML to HTML. Browser-based XSLT Transformation provides very effective separation between the acquisition of the data and the creation of the HTML. The data is passed to the browser and the rules for generating the HTML are kept in a separate XSLT file which the browser fetches from the server and then uses to transform the data. However, the XSLT processing capabilities of different browsers differ markedly. Creating portable stylesheets that can be processed on a large variety of browsers is difficult. In addition, having the browser do all of the XSLT processing is probably the least efficient in terms of time taken to generate the final HTML.
Server-based XSLT Transformation	Mark up the data as XML and then transform it using an XSLT stylesheet and PHP's XSLT processor. Server-based XSLT Transformation is midway between the other two techniques. The separation between the acquisition of the data and the creation of the HTML is maintained but because the XSLT is processed on the server, it is more robust, more efficient and easier to debug.

IMu Web uses Server-based XSLT Transformation. Data (usually retrieved from the IMu server) is marked up into a DOM document (an internal representation of XML). An appropriate XSLT stylesheet is then loaded and applied to the document. Finally, the HTML generated by the transformation is returned to the browser.

We look at this generation of the output in more detail in the next section.

SECTION 5

Generating output

IMu Web requests generate output in several steps:

1. A DOM document is created to represent the data to be returned to the browser as XML.
2. An XSLT stylesheet is loaded.
3. An XSLT processor is created and associated with the stylesheet.
4. The DOM document is transformed using the processor.
5. The generated XHTML is printed.

IMu Web’s search request, which delivers back to the browser an HTML page containing a search screen with a set of tabs, is a good place to see these steps. Here is the entire contents of the `search.php` file:

```
<?php
require_once dirname(__FILE__) . '/request.php';

$class = 'IMuSearchRequest';
class IMuSearchRequest extends IMuWebRequest
{
  public function
  process()
  {
    $xml = $this->newDocument();
    $xml->startElement('search');
    $xml->writeElement('page',
     $this->delParam('page', 'simple'));
    $xml->endDocument();

    $xslt = $this->newStylesheet('search');

    $proc = $this->newProcessor($xslt);

    $html = $proc->transform($xml);
    $html->printXHTML();
  }
}
```

The first lines of the `process` method generate a DOM document. The document is created using `IMuWebRequest`’s `newDocument` method. This method creates a new `IMuWebDocument` object. The `IMuWebDocument` class is itself a subclass of PHP's standard `DOMDocument` class and is described in detail in *Using KE IMu API (With PHP)*.

The document is structured as so:

```
<search>
  <page>simple</page>
</search>
```

The value of the `page` element can be specified using a `page=` parameter in the URL. `IMuWebRequest`'s `delParam` method used here:

```
this->delParam('page', 'simple')
```

removes the `page` parameter from the list of request parameters and returns it. If no `page` parameter was include in the request, the value `simple` is used instead.

Once the document has been constructed the `search` request loads the appropriate stylesheet to be used to transform the XML:

```
$xslt = $this->newStylesheet('search');
```

This method creates a new `IMuWebStylesheet` and then finds and loads the stylesheet whose name is passed as an argument. The `IMuWebStylesheet` class is defined in `common/stylesheet.php`. An XSLT stylesheet is itself an XML document so, appropriately, the `IMuWebStylesheet` class is a DOM document (it is a subclass of `IMuWebDocument`).

The way `IMuWebStylesheet` finds the appropriate stylesheet to load depends on the theme configured by the user. This is described in detail in the next section.

After the stylesheet has been loaded the remaining three steps are quite straightforward:

```
$proc = $this->newProcessor($xslt);

$html = $proc->transform($xml);
$html->printXHTML();
```

The `newProcessor` method creates a new `IMuWebProcessor` and associates the stylesheet with it. `IMuWebProcessor` is a wrapper around the standard PHP `XSLTProcessor` class. (For technical reasons `IMuWebProcessor` is not a subclass of `XSLTProcessor`.)

The processor's `transform` method is then called to process the XML into a new XHTML DOM document. Finally the XHTML is printed to return it to the browser.

SECTION 6

Themes

One of the goals of using XSLT transformations to build the HTML is to allow the same data to be represented in a variety of ways. IMu Web calls these different ways of displaying the data themes.

IMu Web themes are basically a set of XSLT stylesheets, CSS stylesheets, JavaScript files and associated images. Because they are specific to a particular client's Catalogue, the themes are typically stored in the `web/client` directory.

When a user visits an IMu Web page for the first time, IMu Web sets a cookie in the user's browser which contains the name of the theme to use to build the pages. The theme is initially set to the `default-theme` value in the `config.xml` file (see above).

Every time a page is requested, IMu Web gets the theme name from the cookie. Once IMu Web knows which theme is being used it can search for and load the appropriate XSLT stylesheet.

For example, suppose the theme set in the cookie is called `london`. The search request code in the previous section loads a stylesheet called `search` using the `newStylesheet` method:

```
$xslt = $this->newStylesheet('search');
```

This code will try to locate a file called `themes/london/search.xsl`. As in many other cases IMu Web will first look in the `local` directory, then in the `client` directory and then in the `common` directory. For most themes the files are stored under the `client` directory.

XSLT

The stylesheet is a standard XSLT file. The file can be constructed in many ways. It is beyond the scope of this document to describe how to use XSLT. A good XSLT tutorial can be found online at `http://www.w3schools.com/xsl/`.

IMu Web uses a common structure for its XSLT. The following is an example of a preamble commonly used at the start of IMu Web stylesheets:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:imu="http://kesoftware.com/imu">

  <xsl:import href="common.xsl"/>
```

There are a few things to note here:

- The XSLT stylesheets themselves are conventionally encoded in UTF-8. That is, there is no encoding attribute in the `<?xml?>` tag.
- IMu Web stylesheets typically add an `imu` namespace to the document so that tags such as `<imu:img>`, `<imu:string>` and `<imu:submit>` can be included in the XSLT document. These are described in more detail below.
- The stylesheet imports the contents of the `common.xsl` stylesheet. This is a standard way to include common features in different pages. In fact `common.xsl` actually drives the transformation process, calling templates which are actually defined in the other stylesheets.

The theme's `common.xsl` file typically imports a shared `common.xsl` file:

```
<xsl:import href="../shared/common.xsl"/>
```

This means that commonly used templates can be shared across all themes.

It is important to realise that the working directory of the XSLT processor is the directory containing the master stylesheet. Relative path names used in the stylesheet (such as `../shared` in the example above) must be relative to this directory.

The theme's `common.xsl` file also defines the format of the output generated:

```
<xsl:output method="xml" version="1.0"
omit-xml-declaration="yes"/>
```

It may seem a little odd that the output method is set to be xml rather than html. However, IMu Web actually generates XHTML (HTML formatted according to the stricter syntax of XML) and specifying an output of XML causes better XHTML to be created.

How the XSLT is processed

The `common.xsl` file defines a special xsl template which is the starting point of the XSLT processing:

```
<xsl:template match="/">
    <html>
        <xsl:attribute name="dir">
            <xsl:value-of select="$imu-lang-dir"/>
        </xsl:attribute>
        <head>
            <xsl:call-template name="include"/>
            <xsl:call-template name="title"/>
        </head>
        <body>
            <xsl:call-template name="body"/>
        </body>
    </html>
</xsl:template>
```

This template performs five important tasks:

1. It creates the top-level structure for the XHTML page being generated.
2. It sets an `attribute` on the page's top-level `html` element defining the direction that the text used in the page should be displayed. This value is either `ltr` for languages which display text from left to right or `rtl` for languages which display text from right to left. IMu Web uses the two letter language code defined in the configuration file (page 25) to determine whether the value should be `ltr` (most languages) or `rtl` (Arabic and Hebrew).
3. It calls the `include` template. This template is used to include references to CSS stylesheets and JavaScript source code.
4. It calls the `title` template. This template is used to set the title of the page being generated.
5. It calls the `body` template. This template generates the actual page contents.

To understand what happens when each of these templates is called it is important to first understand how XSLT's `import` works. When a stylesheet imports another stylesheet, the templates defined in the first (importing) stylesheet take precedence over those defined in the second (imported) one; we'll call them S1 (importing) and S2 (imported). If both the S1 and S2 define templates with the same name, the template defined in S1 take precedence over those defined in S2. In our example the stylesheet `search.xsl` imports `common.xsl`. If both of these define a template called `body`, the one defined in `search.xsl` will be called rather than the one in `common.xsl`.

This is useful as it allows generic functionality to be defined in the imported stylesheet (S2 or `common.xsl` in our example) which can be overridden in the importing stylesheet (S1 or `search.xsl` in our example) where required. This technique is used throughout the IMu Web stylesheets.

Consider the `title` template. A version of `title` is defined in `common.xsl`. A stylesheet which imports `common.xsl` can choose to use the standard `title` template or to define a new one.

Overriding templates can actually take place at several levels. For example, the version of `body` defined in `common.xsl` simply creates an outer level HTML `div` element and then calls three other templates:

```
<xsl:template name="body">
  <div id="body">
    <xsl:call-template name="header"/>
    <xsl:call-template name="middle"/>
    <xsl:call-template name="footer"/>
  </div>
</xsl:template>
```

Instead of overriding the `body` template a stylesheet which imports `common.xsl` can override any of `header`, `middle` or `footer`. This gives a finer level of control while using as much common functionality as required. It is most common for a stylesheet to override the `middle` template but not override the `header` and `footer` templates. This allows each page to include a common header and footer section around its own content.

The template names (such as `body`) have no special significance. They are named this way to identify their roles in the generation of the page but they have no direct association with any HTML element names.

There is no way of calling a template which has been overridden. This is unfortunate because it means that an overriding template cannot use functionality that is present in the base template. For this reason, IMu Web's base templates which can be overridden call separate templates which actually implement their functionality. For example, the `include` template in `common.xsl` calls another template to generate the HTML which references the standard CSS stylesheets and JavaScript code. These secondary templates follow a naming convention which ensures they do not clash with other templates: the name consists of the theme name followed by the stylesheet name followed by the base template name. For example, here is the general `include` template in `common.xsl` for a theme called `london`:

```
<xsl:template name="include">
  <xsl:call-template name="london-common-include"/>
</xsl:template>
```

Separating the base functionality from the overridden template means that the overriding template can still use the base functionality. For example, here is the `include` template in `search.xsl`, which overrides the one in `common.xsl`:

```
<xsl:template name="include">
  <xsl:call-template name="london-common-include"/>

  <!-- search-specific includes -->
  <link rel="stylesheet" type="text/css"
        href="client/themes/london/search.css"/>
  ...
```

The HTML generated by the stylesheets is reasonably straightforward. However, there are a few conventions that should be understood and followed. These are explained in the following sections.

XHTML

All HTML generated should actually be XHTML. This means, among other things, that all elements must be nested correctly and that all attribute values must be embedded in double quotes. A good description of the differences between standard HTML and XHTML is available online at:

```
http://www.w3schools.com/XHTML/xhtml_html.asp
```

CSS

CSS stylesheets should be used where appropriate. A CSS stylesheet should be named consistently with the XSLT stylesheet that refers to it. For example, IMu Web's Browse pages are built using the XSLT stylesheet `browse.xsl`. CSS styles which are specific to the Browse pages are placed in a file called `browse.css`.

Multilingual Pages

If the pages are designed to be multilingual, the stylesheets themselves should contain no text for display. Instead translations should be added to a `strings.xml` file. This file is typically located in the `client/themes/shared` directory so that translations can be shared between different themes. The file contains translations for each of the languages to be displayed and associates the translations with a logical identifier that can then be used in the XSLT stylesheet.

In this example, each page includes a link at the top labelled `My Museum`. Because this link is to appear at the top of every page it is created in the XSLT `header` template in the `common.xsl` stylesheet. However, because the pages are also to be displayed in French and Arabic the translations must be added to the `strings.xml` file and associated with a logical identifier. The entry in the `strings.xml` file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<strings>
    …
     <string id="common-my-museum">
        <value lang="en">My Museum</value>
        <value lang="fr">Mon Musée</value>
        <value lang="ar">فحتم يدلب</value>
    </string>
    …
</strings>
```

The language of each translation is identified by a two-letter language code. The string id can then be used in the XSLT which generates the HTML:

```
<a>
    …
    <imu:string id="common-my-museum"/>
</a>
```

IMu Web automatically replaces the `<imu:string>` element with the appropriate translation, depending upon which language is being displayed.

If the pages are designed so that they can be displayed in both left-to-right languages and right-to-left languages (as was the case in the previous example), then any CSS styles which specify the horizontal arrangement of elements on the page should be placed in separate stylesheets. For example, any CSS style which specifies the position of a `div` using `float:left` or `float:right` should be placed in a separate stylesheet. Each of the files containing these styles should have `_ltr` or `_rtl` at the end of its name.

For example, IMu Web's Search page displays different search forms as a set of tabs. The tab headings themselves are a set of HTML divs. On a left-to-right page these are displayed floating to the left and on a right-to-left page they should be displayed floating to the right.

The tab headings are created in the `search.xsl` stylesheet and each is given a class of `search-tab`. The corresponding `search.css` stylesheet defines styles for the `search-tab` class:

```
.search-tab
{
    background-color: #484848;
    color: #FFFFFF;
    cursor: pointer;
    font-weight: bold;
    padding-top: 3px;
}
```

These styles apply regardless of whether the page is displayed left-to-right or right-to-left. Some other styles differ depending on the orientation of the page. Styles which should only be applied when the page is being displayed left-to-right are defined in the `search_ltr.css` file:

```
.search-tab
{
    float: left;
    margin-left: 0px;
    margin-right: 10px;
    padding-left: 3px;
}
```

Styles which should only be applied when the page is being displayed right-to-left are defined in the `search_rtl.css` file:

```
.search-tab
{
    float: right;
    margin-left: 10px;
    margin-right: 0px;
    padding-right: 3px;
}
```

IMu Web determines the direction to be displayed based on the language and sets a global XSLT variable, `$imu-lang-dir`, which contains the value `ltr` or `rtl`. The stylesheet then includes the appropriate file in the include template:

```
<xsl:template name="include">
    …
    <link rel="stylesheet" type="text/css"
            href="client/themes/london/search.css"/>
    <link rel="stylesheet" type="text/css">
        <xsl:attribute name="href">
            <xsl:text>client/themes/london/search_</xsl:text>
            <xsl:value-of select="$imu-lang-dir"/>
            <xsl:text>.css</xsl:text>
        </xsl:attribute>
    </link>
    …
</xsl:template>
```

JavaScript

IMu Web pages are typically designed to make use of JavaScript if it is available but not to be entirely reliant on it. The general design philosophy of IMu Web is to build pages which operate successfully without JavaScript but allow JavaScript to be used to enhance the pages.

Areas where JavaScript is used within IMu Web are:

- Show search results as part of the search page, rather than in a separate page
- Add transition effects such as scrolling or fade-in when moving from one results page to another.
- Provide "pre-searching" such as automatically marking up terms in a page (such as a list of subjects) with the count of the number of records which match these terms.
- Simplifying user input by acting immediately when a user chooses a value from a drop-down list or selects a checkbox. This removes the need for the user to click a Go or Update button.

None of these is essential to the operation of the pages; they simply "add value".

There are three reasons for designing the standard IMu Web pages this way:

1. It may be a requirement of some organisations that their website can operate without JavaScript. This is becoming less and less common but it does still happen.
2. Even if the organisation does not require this, a user may choose to disable JavaScript in their browser. Similarly, some browsers running on mobile devices do not always implement JavaScript thoroughly, although this is changing rapidly.
3. The work of debugging the construction of pages during development is made much easier if the page can be built and displayed without using JavaScript.

SECTION 7

Browsing

IMu Web provides a set of browse pages which allow users to view and navigate their way through a hierarchical set of EMu Narrative records.

The browse pages are generated by a request of the form:

```
http://web-server/path/imu.php?request=browse
```

This request displays a main narrative record and a summary of its sub-narratives.

There is one additional parameter which may be passed to the request:

`irn`	The irn of the record to display as the main narrative. If this parameter is not used, the master narrative is displayed. The narrative to be identified as the master narrative may be configured (see below).

The browse request is handled by an `IMuBrowseRequest` object. This is defined in `common/browse.php`.

The `process` method of `IMuBrowseRequest` loads configuration information from a configuration file, `browse.xml`. A default `browse.xml` configuration file is distributed with IMu Web in the `common` directory but it may be overridden by a `browse.xml` file in the `local` or `client` directory as required.

The configuration file can be used to specify:

1. The record to be identified as the master (i.e. top-level) narrative.
2. The set of columns to be retrieved for each narrative.

The record to be identified as the master narrative is configured as follows:

```
<?xml version="1.0"?>
<browse>
    …
    <master column="DesType_tab">master</master>
    …
</browse>
```

This specifies a search for the word master in the `DesType_tab` column. This is the default defined in the `common/browse.xml` file. Both the column to be searched and the value to search for can be changed. For example, the master narrative can be set by its `irn`:

```
    <master column="irn">42</master>
```

If the `IMuBrowseRequest` object does not find exactly one matching narrative which can act as the master, it generates an error.

The set of columns to be retrieved for each narrative is configured in the `<columns>` section of the `config.xml` file as follows:

```
<?xml version="1.0"?>
<browse>
    …
    <columns>
        <column>irn</column>
        <column>NarTitle</column>
        <column>NarNarrative{format:html-clean}</column>
        <column>image</column>
        <column>
            children=HieChildNarrativesRef_tab.
            (
                irn,
                NarTitle,
                NarNarrative{format:html-clean},
                image
            )
        </column>
        <column>
            trails.
            (
                irn,
                NarTitle
            )
        </column>
    </columns>
    …
</browse>
```

Each column to be fetched from the server is specified in a `<column>` element. The format of the column specifications is described in detail in *Using KE IMu API (With PHP).*

The column definitions shown in this example are the default ones from `common/browse.xml`. This requests that the server provides:

1. The irn of the main narrative.
2. The title of the main narrative (`NarTitle`).
3. The text of the main narrative (`NarNarrative`). This column itself contains HTML. The `{format:html-clean}` directive that follows the column name requests that the server reformats the HTML before returning it. The reformatting serves two purposes:
    - It ensures that the HTML in the column is actually reformatted as valid XHTML. This is essential to ensure that the HTML can be included in the overall XML document generated by the request.
    - It removes any specific font changes in the HTML which can conflict with the web pages' own styles.
4. The image associated with the narrative. The `image` column is a virtual column and does not actually exist in the enarratives table. When a request is made for the `image` column, the server looks up the list of multimedia attached to the record and finds the first which is an image (as opposed to some other form of multimedia such as document, audio or video) with the appropriate security allowing it to be displayed.

5. The set of breadcrumb trails which indicate the position of this narrative in the overall narrative hierarchy. The `trails` column is a virtual column and does not actually exist in the enarratives table. When a request is made for the `trails` column, the server finds the complete set of paths through the narrative hierarchy from the master (top-level) narrative to the current narrative based on attachments in the `HieChildNarrativesRef_tab` column. For each breadcrumb in each trail the irn and the narrative title are returned.
6. A list of all the main narrative's "children" (or sub-narratives). The child narratives are stored as attachments in the `HieChildNarrativesRef_tab` column. For each child narrative the server returns its irn, title, the narrative text (also reformatted) and its associated image.

As described in previous sections, the request retrieves the data from the IMu server, marks it up as XML and then uses theme-based XSLT stylesheets to generate the XHTML.

The XML generated is structured as follows:

```
<?xml version="1.0"?>
<result>
  <hits>1</hits>
  <rows>
    <row>
      <rownum>1</rownum>
      <irn>11</irn>
      <NarTitle>Fine Art &amp; Sculpture</NarTitle>
      <NarNarrative><p>One of the nation's great artistic and
historic resources, the National Museum's Fine Art &amp; Sculpture
collection includes over 10,000 items - objects and virtual
displays. The collection spans the range of artistic media
from&#xA0;paintings, sculpture,&#xA0;ceramics and tapestries from
the High Renaissance, Mannerist, Baroque, Rococo, and Neoclassical
periods through to the new digital artforms of the&#xA0;present
day.</p></NarNarrative>
      <image>
        <irn>7228</irn>
        <format>tiff</format>
        <type>image</type>
      </image>
      <trails>
        <trail>
          <entry>
            <NarTitle>The Museum's Collections</NarTitle>
            <irn>19</irn>
          </entry>
        </trail>
      </trails>
      <children>
        <child>
          <irn>2</irn>
          <NarTitle>Portrait of William Wilberforce</NarTitle>
          <NarNarrative><div><p>This portrait …</NarNarrative>
          <image>
            <irn>7235</irn>
            <format>jpeg</format>
            <type>image</type>
          </image>
        </child>
        <child>
          …
        </child>
        …
      </children>
    </row>
  </rows>
</result>
```

A theme’s XSLT stylesheet can process this XML in any way it chooses.

Example: IMU Web's london theme

IMu Web’s standard `london` theme processes this structure as follows:

1. It imports the `common.xsl` stylesheet.

   This means that the `browse.xsl` is processed as described above:

   ```
   <xsl:import href="common.xsl"/>
   ```
2. It overrides the `include` template to add in references to its own CSS stylesheets (both the standard one, `browse.xsl`, and one for the appropriate text direction, `browse_ltr.css` or `browse_rtl.css`):

   ```
   <xsl:template name="include">
       <xsl:call-template name="london-common-include"/>

       <link rel="stylesheet" type="text/css"
           href="client/themes/london/browse.css"/>
       <link rel="stylesheet" type="text/css">
           <xsl:attribute name="href">
               <xsl:text>client/themes/london/browse_</xsl:text>
               <xsl:value-of select="$imu-lang-dir"/>
               <xsl:text>.css</xsl:text>
           </xsl:attribute>
       </link>
       <script type="text/javascript"
           src="client/themes/london/browse.js"/>
   </xsl:template>
   ```
3. It overrides the `title` template to set the title on the browser window to the narrative title:

   ```
   <xsl:template name="title">
       <title>
           <xsl:value-of select="result/rows/row/NarTitle"/>
       </title>
   </xsl:template>
   ```
4. It overrides the `middle` template to create the content in the page. Notice that it does not override the `body` template, meaning that the standard `header` and `footer` templates defined in `common.xsl` will be used:

   ```
   <xsl:template name="middle">
       <div id="browse">
           <xsl:apply-templates select="result/rows/row"/>
       </div>
   </xsl:template>
   ```
5. The `middle` template applies a template which matches each `row` element in the data.

   As each Browse page shows one narrative record there will be only one matching record.

6. The matching template first displays a drop-down list to the right of the page:

```
<xsl:template match="result/rows/row">
    <table id="browse-collection">
        <tr>
            <td style="width:99%"/>
            <td style="width:1%">
                <xsl:call-template
                    name="london-common-collection"/>
            </td>
        </tr>
    </table>
    …
</xsl:template>
```

   This allows users to select which of their personal collections they want to be able to add selected records to. The drop-down list is built using the common template called `london-common-collection`. For more details regarding collections refer to *Collections* (page 71).

You will notice that the checkbox is positioned to the right of the screen using a table. IMu Web uses tables in several places to control the positioning of elements on the page. This is often considered extremely poor design and vast ideological battles have been fought in cyberspace over the use of tables for layout in web pages. Proponents of table-less page layouts extol the virtues of using floating divs under sophisticated CSS control.

Pages generated by IMu Web need to be extremely portable. They need to display correctly on a wide variety of browsers offering significantly different levels of HTML and especially CSS compatibility. They need to be capable of being displayed with different screen resolutions and font sizes on both conventional displays and mobile devices. They also need to operate correctly when displayed in both left-to-right and right-to-left mode.

Additionally, most standard IMu Web pages are designed to use as much of the screen as possible, rather than be limited to a fixed size which is severely compromised on wide displays. Coupled with this is the requirement that, with very few exceptions, the CSS styles used must not specify the size of elements in absolute sizes, as this is inherently not portable between browsers and devices and results in areas of the display that do not resize according to their content.

Currently, the combination of all these requirements makes it extremely difficult to layout pages without sometimes using tables. There are situations where a layout generated using floating divs alone simply cannot meet our requirements. This situation may change with increased standardisation of browsers but currently there is no sign of it.

This does not, of course, mean that you should not use table-less layouts, rather that you should be aware that every web page design involves some form of compromise.

7. The template then displays the breadcrumb trails:

```
<div id="browse-trails">
    <xsl:for-each select="trails/trail">
        <div>
            <xsl:for-each select="entry">
                <xsl:if test="position() &gt; 1">
                    <xsl:text>&#160;&gt;&#160;</xsl:text>
                </xsl:if>
                <a>
                    <xsl:attribute name="href">
                        <xsl:value-of select="$imu-request"/>
                        <xsl:text>&amp;irn=</xsl:text>
                        <xsl:value-of select="irn"/>
                    </xsl:attribute>
                    <xsl:value-of select="NarTitle"/>
                </a>
            </xsl:for-each>
        </div>
    </xsl:for-each>
</div>
```

   There may be several trails to the narrative. Each is displayed in a separate `div`. Each crumb on the trail is displayed using the narrative's title and is marked up as a hyperlink. The link's `href` attribute is constructed using the global XSLT processor variable `$imu-request`. This variable is created by the PHP `IMuWebProcessor` object and contains the URL of the request, including the `request=` parameter but without any additional parameters. In other words it contains text like this:

   `http://`*web-server*`/`*path*`/imu.php?request=browse`

8. The main part of the page is built into a form:

```
<form>
    <input type="hidden" name="request" value="group"/>
    <input type="hidden" name="action" value="update"/>

    <xsl:call-template name="browse-main"/>
    <xsl:call-template name="browse-children"/>

    <imu:submit class="group-update" value="common-update"/>
</form>
```

   The purpose of the form is to allow users to select narratives in which they are interested and add them to their collections. Remember that the pages are designed to operate without requiring JavaScript (page 40).

   The form includes two hidden `input` elements. The `request` element is set to `group`. Group requests are used to add or remove records from a user's collection (internally known as a group).

9. The body of the `form` element is constructed by calling two templates, `browse-main` and `browse-children`. Not surprisingly these are used to build the HTML to display the main narrative and the child narratives respectively.

10. The standard form Submit button is created with an `<imu:submit>` element. This is used for multi-lingual pages, similar to the way the `<imu:string>` element is used. IMu Web interprets the `value` attribute as a string id - see *Multilingual Pages* (page 38) for details - finds the appropriate translation for the button label and then automatically replaces this element with the standard `<input type="submit">` form element.

The `browse-main` and `browse-children` templates are fairly straightforward XSLT templates and will not be described in detail. The only complexity in the templates is the provision of a checkbox associated with each narrative (either the main narrative or a child) which allows the user to add the narrative to (or remove it from) one of their collections. This is explained in more detail in *Collections* (page 71).

The browse pages include some JavaScript. The JavaScript uses JQuery, a third-party JavaScript library (see http://jquery.com/) to install event handlers into each checkbox so that when it is clicked, the user's collection will automatically be updated without requiring the user to click the Update button.

SECTION 8

Search and Display

IMu Web provides a set of search and display pages. These pages allow the user to search for matching records and view the results.

Search Page

The search pages are generated by a request of the form:

`http://`*web-server*`/`*path*`/imu.php?request=search`

This request displays a set of search tabs with the Simple search page shown.

There is one additional parameter which may be passed to the request:

`page`	The name of the search page to be displayed. If this parameter is not used, the Simple search page is displayed.

The search request is handled by an `IMuSearchRequest` object. This is defined in `common/search.php`. It generates a very simple XML structure:

```
<?xml version="1.0"?>
<search>
    <page>simple</page>
</search>
```

The XSLT in the standard `london` theme is quite straightforward. As was the case with the stylesheet for the Browse pages, the `search.xsl` stylesheet implements the `include`, `title` and `middle` templates.

The `include` template includes both CSS stylesheets and JavaScript specific to the Search pages. It also includes CSS stylesheets and JavaScript for the display pages:

```
<xsl:template name="include">
    <xsl:call-template name="london-common-include"/>

    <link rel="stylesheet" type="text/css"
        href="client/themes/london/search.css"/>
    <link rel="stylesheet" type="text/css">
        <xsl:attribute name="href">
            <xsl:text>client/themes/london/search_</xsl:text>
            <xsl:value-of select="$imu-lang-dir"/>
            <xsl:text>.css</xsl:text>
        </xsl:attribute>
    </link>
    <script type="text/javascript"
        src="client/themes/london/search.js"/>

    <link rel="stylesheet" type="text/css"
        href="client/themes/london/display.css"/>
    <link rel="stylesheet" type="text/css">
        <xsl:attribute name="href">
            <xsl:text>client/themes/london/display_</xsl:text>
            <xsl:value-of select="$imu-lang-dir"/>
            <xsl:text>.css</xsl:text>
        </xsl:attribute>
    </link>
    <script type="text/javascript"
        src="client/themes/london/display.js"/>
    <script type="text/javascript"
        src="client/themes/london/display/details.js"/>
    <script type="text/javascript"
        src="client/themes/london/display/lightbox.js"/>
    <script type="text/javascript"
        src="client/themes/london/display/list.js"/>
</xsl:template>
```

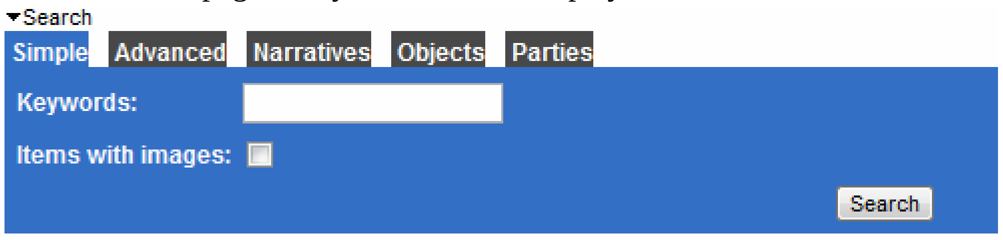
The display CSS and JavaScript are included in the search page because IMu Web can use JavaScript to insert the display results into the search page. The means that CSS stylesheets and JavaScripts used by the display pages need to be included in the search page itself.

The `middle` template is straightforward:

```
<xsl:template name="middle">
    <div id="search">
        <div id="search-prompt">
            <imu:img src="arrow-down.png"/>
            <imu:string id="search-prompt"/>
        </div>
        <div class="search-area">
            <div id="search-tabs">
                <xsl:call-template name="simple-tab"/>
                <xsl:call-template name="advanced-tab"/>
                …
            </div>
            <div id="search-pages">
                <xsl:call-template name="simple-page"/>
                <xsl:call-template name="advanced-page"/>
                …
            </div>
        </div>
    </div>
</xsl:template>
```

This creates:

1. A div which contains a Search prompt, preceded by a down arrow.
2. A tab area, showing the name of each search page.
3. A set of search pages, only one of which displays:



If JavaScript is being used, the down arrow next to the Search prompt can be clicked to hide the entire search area.

The tabs are a set of floating divs. Here is the XSLT that creates the Simple tab:

```
<xsl:template name="simple-tab">
    <div page="simple">
        <xsl:attribute name="class">
            <xsl:text>search-tab</xsl:text>
            <xsl:if test="search/page = 'simple'">
                <xsl:text> search-tab-active</xsl:text>
            </xsl:if>
        </xsl:attribute>
        <a>
            <xsl:attribute name="href">
                <xsl:value-of select="$imu-request"/>
                <xsl:text>&amp;page=simple</xsl:text>
            </xsl:attribute>
            <imu:string id="search-simple"/>
        </a>
    </div>
</xsl:template>
```

Each tab is a `div` containing a single hyperlink used to select a different search page. As usual, the page is made multilingual by using an `<imu:string>` element for the tab label. The tabs are coloured differently by having the CSS class `search-tab-active` added to the `div` if it is the selected tab. Here is the relevant section from the `search.css` file:

```
.search-tab
{
    background-color: #484848;
    color: #FFFFFF;
    cursor: pointer;
    font-weight: bold;
    padding-top: 3px;
}

.search-tab-active
{
    background-color: #336FC5;
}

.search-tab a
{
    color: #FFFFFF;
    text-decoration: none;
}
```

The styles which float the divs, either to the left or the right, are in separate stylesheets. This is described in *Multilingual Pages* (page 38).

The search pages are built in a similar fashion. Each search page is a div containing a form. Here is the XSLT that builds the Simple search page:

```
<xsl:template name="simple-page">
    <div page="simple">
        <xsl:attribute name="class">
            <xsl:text>search-page</xsl:text>
            <xsl:if test="search/page = 'simple'">
                <xsl:text> search-page-active</xsl:text>
            </xsl:if>
        </xsl:attribute>
        <form>
            <input type="hidden" name="request" value="load"/>
            <table>
                <tr>
                    <td class="search-prompt">
                        <imu:string id="search-field-keywords"/>
                        <xsl:text>:</xsl:text>
                    </td>
                    <td>
                        <input type="text" name="keywords"/>
                    </td>
                </tr>
                <tr>
                    <td class="search-prompt">
                        <imu:string id="search-field-images"/>
                        <xsl:text>:</xsl:text>
                    </td>
                    <td>
                        <input type="checkbox"
                            name="MulHasMultiMedia"
                            value="Y"/>
                    </td>
                </tr>
            </table>
            <imu:submit value="search-button"/>
        </form>
    </div>
</xsl:template>
```

There are several things to note here. All pages comprise a `div` which has a CSS class of `search-page`. The selected page also has a class of `search-page-active`. In the `search.css` CSS stylesheet the `search-page` class is defined to make the `div` invisible while the `search-page-active` class overrides this and makes the selected page visible:

```
.search-page
{
    display: none;
    …
}

.search-page-active
{
    display: block;
}
```

The form has a hidden `input` element. This ensures that the form submits a request such as:

```
http://web-server/path/imu.php?request=load
```

The actual form prompts, such as "Keywords" and "Items with images", are added to the form using `<imu:string>` elements.

The form `input` elements are given names that will be passed directly to the `load` request when the form is submitted. Each name may be either the name of an actual column in an EMu table or a search alias. In the case of Simple search page shown above the "Items with images" checkbox, the name (`HasMultimedia`) corresponds directly to a column in an EMu table. The "Keywords" textbox has a name (`keywords`) which is a search alias. Search aliases are explained in more detail below.

As with the Browse pages, each form's submit button is added to the page using an `<imu:submit>` element.

Running a Search

When a search form is completed and submitted, the page generates an IMu Web `load` request. The `load` request is responsible for executing the search and generating a result set.

The `load` request is handled by an `IMuLoadRequest` object. This is defined in `common/load.php`.

The `process` method of `IMuLoadRequest` loads configuration information from a configuration file, `modules.xml`, which is usually located in the `client` directory.

The configuration file can be used to specify:

1. The set of EMu modules to be searched and the order in which they are to be searched.
2. Search aliases.
3. Record views.

Set of EMu modules

The set of EMu modules to be searched is configured in the `include` section of the confirmation file as follows:

```
<?xml version="1.0"?>
<modules>
    …
    <include>
        <module>enarratives</module>
        <module>ecatalogue</module>
        <module>eparties</module>
    </include>
    …
</modules>
```

The order in which the modules are to be searched is defined by the order they are listed in the `include` section.

This list represents the full set of modules to be searched. A request can explicitly reduce the number of modules to be searched by passing a parameter:

```
http://web-server/path/imu.php?request=load...&ecatalogue=off...
```

This will mean that this `load` request will not include the ecatalogue module in the set of modules it searches.

Search alias

A search alias associates a set of columns in different modules with a logical name. Search aliases are defined in the `modules.xml` file as follows:

```
<?xml version="1.0"?>
<modules>
    …
    <aliases>
        …
        <alias id="keywords">
            <module id="enarratives">
                <column>DesSubjects_tab</column>
                <column>NarNarrative</column>
                <column>NarTitle</column>
                <column>SummaryData</column>
            </module>
            <module id="ecatalogue">
                <column>CreSubjectClassification_tab</column>
                <column>SummaryData</column>
            </module>
            <module id="eparties">
                <column>BioCommencementNotes</column>
                <column>SummaryData</column>
            </module>
        </alias>
        …
    </aliases>
    …
</modules>
```

This entry creates an alias called "keywords" and associates it with a different set of columns to be searched in each of three different modules. The effect of the alias is that when a search term such as `keywords=design` is submitted to the `load` request the search which is actually executed by the server is an OR search across different columns in the different modules being searched. This is a powerful feature of IMu Web as it allows generic search forms, such as the Simple form described above, to be built easily.

Record view

A record view is similar to a search alias in that it associates a set of columns in different modules with a logical name. Views are defined as follows:

```
<?xml version="1.0"?>
<modules>
    …
    <views>
        <defaults>
            <module id="enarratives">
                <column>NarTitle</column>
                …
            </module>
            <module id="ecatalogue">
                <column>SummaryData</column>
                …
            </module>
            <module id="eparties">
                <column>NamPartyType</column>
                …
            </module>
        </defaults>

        <view id="details">
            <count>1</count>
            <module id="enarratives">
                <column>DesSubjects_tab</column>
                <column>DesType_tab</column>
                …
            </module>
            <module id="ecatalogue">
                <column>AssOtherAssociationNotes0</column>
                <column>CreDateCreated</column>
                …
            </module>
            <module id="eparties">
                <column>AddPhysCity</column>
                <column>AddPhysCountry</column>                …
                …
            </module>
        </view>
        …
        <view id="list">
            <count>6</count>
            <module id="enarratives">
                <column>NarNarrative{format:html-strip}</column>
                …
            </module>
            <module id="ecatalogue">
                <column>PhyDescription</column>
                <column>CreDateCreated</column>
                …
            </module>
            <module id="eparties"/>
        </view>
    </views>
    …
</modules>
```

These entries create two views, one called `details` and one called `list`. Each view defines two pieces of information:

- The default number of records to be fetched (specified in the `<count>` element).
- The list of columns to be fetched from each module.

The `<defaults>` section is used to specify additional columns to be fetched for each module, regardless of which view is being used. This is provided for convenience to save having to list the same columns in each view.

The set of columns defined for a view represents what the lower level IMu API calls a Fetch Set. Fetch Sets, and their use when searching across several modules (as IMu Web does), are described in more detail in *Using KE IMu API (With PHP)*.

The load request

The `load` request is used to create a result set containing matching records from several modules. Typically this is done by executing a search in which case the parameters passed to the request are usually either search alias names or EMu column names.

For example, a simple search using a `keywords` alias can be run using the URL:

```
http://.../imu.php?request=load&keywords=design
```

When preparing for a search the `load` request looks for two special types of parameters:

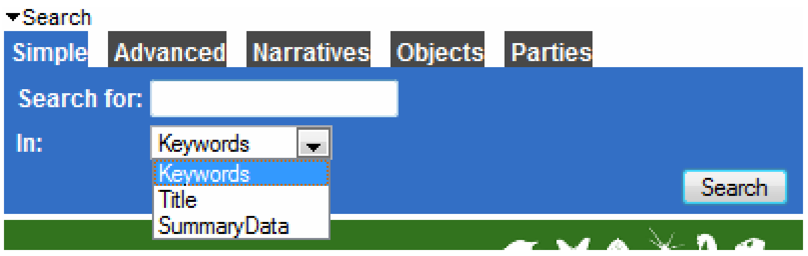
- `column`
- `value`

These can be used to specify the column (or alias) for the search and the value to search for. The request can include any number of these parameters, `column1`/`value1`, `column2`/`value2` and so on.

The previous search can also be run using a URL like this:

```
http://.../imu.php?request=load&column1=keywords&value1=design
```

This method of specifying a search allows you to build search forms where the user can select which fields are to be searched. For example:



In this example the text box is given the name `value1` and the drop-down list is given the value `column1`. Here is the HTML fragment to build this form:

```
<form>
  …
  <input type="hidden" name="request" value="load"/>
  …
  <table>
    <tr>
      <td class="search-prompt">Search for:</td>
      <td>
        <input type="text" name="value1"/>
      </td>
    </tr>
    <tr>
      <td class="search-prompt">In:</td>
      <td>
        <select name="column1">
          <option name="keywords">Keywords</option>
          <option name="title">Title</option>
          <option name="SummaryData">Summary Data</option>
        </select>
      </td>
    </tr>
    …
  <table>
  …
</form>
```

In addition to running a search, the `load` request can also create a result set by restoring a set of records saved in a collection. This is described in more detail in *My Museum* (page 75).

Once the `load` request has created the result set it then redirects the browser to a `display` request to display the first set of results.

Displaying Results

Pages of results are built by the `display` request.

The `display` request requires two parameters:

`port`	The number of the port to be used to re-connect to the IMu server.
`id`	The id of the server-side handler created by the `load` request.

Both of these parameters are needed to allow the `display` request to use the result set created by the `load` request. The role of the `port` and `id` parameters is described in detail in the Maintain State section of *Using KE IMu API (With PHP)*.

There are several additional parameters which may be passed to the request:

`flag` `offset`	The `flag` and `offset` parameters define the starting position of the block of records in the result set to be used to build the page. The `flag` parameter is a string and can be either one of `start`, `current` or `end` or the name of one of the modules listed in the `include` section of `modules.xml`. If a `flag` parameter is not included in the request, it defaults to `start`. The `offset` parameter is an integer. It adjusts the starting position relative to the `flag` parameter. 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`. If an `offset` parameter is not included in the request, it defaults to `0`.
`view`	The name of the view to be used. This must be the name of one of the views defined in the `views` section of `modules.xml`. If a `view` parameter is not included in the request, it defaults to the `<default-view>` value defined in the `config.xml` file. This defines the set of columns to be fetched from the server for each record.
`count`	The number of records to be fetched to build the page. If this parameter is not included in the request, or is the word `default`, it defaults to the value defined in the `<count>` element for the chosen view.
`format`	The format of the data the request should generate. This must be one of: `html`, `json`, or `xml`. If this parameter is not included in the request, it defaults to `html`.

The `display` request connects to the server on the supplied `port` and uses the `id` to identify the result set it wants to use. It then fetches the data in the columns specified by the `view` parameter for the set of records defined by the `flag`, `offset` and `count` parameters. Once the data has been fetched by the server the request processes the data and outputs it in the required format.

If the output format requested is `json` or `xml`, the processing of the data is straightforward. If the format is `json`, the data returned by the server is simply marked up in JavaScript Object Notation (JSON) and delivered to the browser. If the format is `xml`, the `display` request builds a DOM document from the data returned by the server and then generates an XML document from that.

If the output format requested is `html`, the data from the server is processed in the same way as described for the Browse pages above. The data is marked up as XML and then transformed using theme-based XSLT stylesheets to generate the XHTML.

The XML created is structured as follows:

```
<?xml version="1.0"?>
<result>
  <count>6</count>
  <modules>
    <module>
      <name>enarratives</name>
      <index>0</index>
      <hits>16</hits>
      <rows>
        <row>
          <rownum>15</rownum>
          <irn>50</irn>
          <NarTitle>Face and Place…</NarTitle>
          <NarNarrative>A dramatic growth in…</NarNarrative>
          …
        </row>
        <row>
          <rownum>16</rownum>
          <irn>52</irn>
          <NarTitle>…</NarTitle>
          <NarNarrative>…</NarNarrative>
          …
        </row>
      </rows>
    </module>
    <module>
      <name>ecatalogue</name>
      <index>1</index>
      <hits>7</hits>
      <rows>
        <row>
          <rownum>1</rownum>
          <irn>100037</irn>
          <TitMainTitle>Dress from …</TitMainTitle>
        </row>
        <row>
          <rownum>2</rownum>
          <irn>107939</irn>
          <TitMainTitle>The Maestro's Company</TitMainTitle>
        </row>
        <row>
          <rownum>3</rownum>
          …
        </row>
        <row>
          <rownum>4</rownum>
          …
        </row>
      </rows>
    </module>
  </modules>
  <current>
    <flag>enarratives</flag>
    <offset>14</offset>
  </current>
  <prev>
    <flag>enarratives</flag>
    <offset>8</offset>
  </prev>
  <next>
    <flag>ecatalogue</flag>
    <offset>4</offset>
  </next>
  <links>
    <link>
      <module>enarratives</module>
      <url>/artdemo/imu.php?request=display&amp;port=…</url>
    </link>
    <link>
      <module>ecatalogue</module>
      <url>/artdemo/imu.php?request=display&amp;port=…</url>
    </link>
    <link>
      <module>eparties</module>
      <url>/artdemo/imu.php?request=display&amp;port=…</url>
    </link>
  </links>
</result>
```

The `<count>` element indicates the number of records fetched. The records retrieved are grouped by module in the `<modules>` section. In this example six records were retrieved. The first two records came from the Narratives module and the remaining four came from the Catalogue module. This means that the `<modules>` section contains two `<module>` sections, one for Narratives (enarratives) and one for the Catalogue (ecatalogue).

Each `<module>` section includes the following elements:

`<name>`	The name of the module (enarratives, ecatalogue, etc.).
`<index>`	The position of the module in the list of modules in the `<include>` section of the `modules.xml` file.
`<hits>`	An estimate of the number of matching records for this module in the result set. This is the same as the information returned by the `browse` request.
`<rows>`	The data for each record.

This XML is processed in a similar way to the Browse pages described in the previous section. The `display` request loads a stylesheet specific to the view selected:

```
$xslt = $this->newStylesheet('display/' . $result->view);
```

For example, if the `list` view is selected, the main stylesheet used to process the XML will be `display/list.xsl`. The stylesheet for each specific view imports a more general `display.xsl` and a separate stylesheet for each module which may be displayed:

```
<xsl:import href="../display.xsl"/>
<xsl:import href="list-ecatalogue.xsl"/>
<xsl:import href="list-enarratives.xsl"/>
<xsl:import href="list-eparties.xsl"/>
```

The `display.xsl` stylesheet in turn imports `common.xsl` in the same way as `browse.xsl` and then overrides the `middle` template:

```
<xsl:template name="middle">
    <div class="display">
        <table class="display-toolbar">
            …
        </table>

        <div class="display-region">
            <div class="display-content">
                <xsl:attribute name="url">
                    <xsl:value-of select="$imu-url"/>
                </xsl:attribute>
                <xsl:call-template name="display-content"/>
            </div>
        </div>
    </div>
</xsl:template>
```

The middle template builds a framework common to all views. The framework consists of an outer `div` with a class attribute set to display and two inner divs. The class attributes are important as they are used by JavaScript code when implementing transition effects such as scrolling or fading from one page to the next.

The middle template then calls the display-content template. This template implements a specific view and is, not surprisingly, defined in the view-specific stylesheet.

Example: List view

By way of example, here is how the list view is built.

The stylesheet is in the `display/list.xsl` file. It implements the `display-content template` as follows:

```
<xsl:template name="display-content">
    <xsl:call-template name="london-display-navigation"/>
    <form>
        <input type="hidden" name="request" value="group"/>
        <input type="hidden" name="action" value="update"/>
        <div class="display-area">
            <div class="display-page">
                <xsl:call-template name="list-page"/>
            </div>
        </div>
        <imu:submit class="group-update" value="common-update"/>
    </form>
</xsl:template>
```

The template builds a form which allows users to add records in which they are interested to their collections. It then calls a template which generates output for each record to be displayed:

```
<xsl:template name="list-page">
    <table class="list-table">
        <xsl:for-each select="result/modules/module/rows/row">
            <tr>
                <xsl:call-template name="list-row"/>
            </tr>
        </xsl:for-each>
    </table>
</xsl:template>
```

The `list-row` template constructs an HTML framework for each record and finally calls a template that is specific to the module from which the record comes:

```
<xsl:template name="list-row">
    …
    <td style="width:99%">
        <div class="list-content">
            <xsl:choose>
                <xsl:when test="../../name = 'ecatalogue'">
                    <xsl:call-template name="list-ecatalogue"/>
                </xsl:when>
                <xsl:when test="../../name = 'enarratives'">
                    <xsl:call-template name="list-enarratives"/>
                </xsl:when>
                <xsl:when test="../../name = 'eparties'">
                    <xsl:call-template name="list-eparties"/>
                </xsl:when>
            </xsl:choose>
        </div>
    </td>
    …
</xsl:template>
```

The module-specific templates are defined in the modules specific stylesheets imported at the top of `display/list.xsl`. For example, the `list-enarratives` template is defined in `display/list-enarratives.xsl`. These templates generate the actual content. For example, the list view of a Narrative record is generated as follows:

```
<xsl:template name="list-enarratives">
    <table class="list-header">
        <tr>
            <td class="list-title">
                <a>
                    <xsl:attribute name="href">
                        <xsl:value-of select="$imu-display"/>
                        <xsl:text>&amp;flag=</xsl:text>
                        <xsl:value-of select="../../name"/>
                        <xsl:text>&amp;offset=</xsl:text>
                        <xsl:value-of select="rownum - 1"/>
                        <xsl:text>&amp;count=default</xsl:text>
                        <xsl:text>&amp;view=details</xsl:text>
                    </xsl:attribute>
                    <xsl:value-of select="NarTitle"/>
                </a>
            </td>
            <td class="list-select">
               <xsl:call-template name="london-display-checkbox"/>
            </td>
        </tr>
    </table>
    <xsl:value-of select="NarNarrative"/>
</xsl:template>
```

This includes three components: the narrative title, the narrative text and a checkbox (used for adding the record to a collection). The title is marked up as a link to display the narrative record details.

The XSLT to generate other views (e.g. lightbox and details) uses the same technique.

JavaScript

The search and display pages can operate without the need for JavaScript. However, where JavaScript is available these pages will make use of it. JavaScript is used to:

- Add a record to a collection without requiring the user to click the Update button.
- Allow the user to hide the search forms.
- Switch between search tabs without having to resubmit the page.
- Incorporate the search results into the search page rather than displaying on a separate page.
- Provide special transition effects when moving from one results page to another.

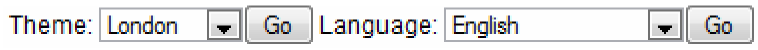
SECTION 9

User Preferences

When a user requests any IMu Web page, the base request handler checks whether a cookie has been set in the user's browser. By default the cookie is called `IMu` (although it can be changed in `config.xml`, as described above). If the cookie does not exist, the request handler assumes that this is the user's first visit to the website. The handler creates a new record for the user in the EMu `ewebusers` table. This record stores the user's IP address, their preferred language and theme and whether they want to use JavaScript or not. These values are initially set to the defaults defined in `config.xml` (page 25).

The request then stores information back into the cookie. Most importantly it stores the irn of the `ewebusers` records for that user so that it will be able to find the information about the user again. When the cookie is stored it is given a duration which controls how long until it is no longer valid and will not be submitted by the browser along with a request. By default the cookie is valid for 7 days. The cookie's expiry date is updated every time an IMu Web request is submitted.

The standard IMu Web pages allow the user to change the theme and the language to be used from any page. For example, the `london` theme includes these drop down lists at the top of each page:



The page contains two HTML forms. The XSLT to generate these forms is found in the `header` template in the `common.xsl` stylesheet:

```
<xsl:template name="header">
  <table id="header-top">
    <tr>
      …
      <td id="header-theme">
        <form style="display:inline">
          <input type="hidden" name="request" value="config"/>
          <imu:string id="common-theme"/>
          <xsl:text>: </xsl:text>
          <imu:select type="themes" name="theme"/>
          <imu:submit value="common-go"/>
        </form>
      </td>
      <td id="header-language">
        <form style="display:inline">
          <input type="hidden" name="request" value="config"/>
          <imu:string id="common-language"/>
          <xsl:text>: </xsl:text>
          <xsl:call-template name="london-common-language"/>
          <imu:submit value="common-go"/>
        </form>
      </td>
    </tr>
  </table>
  …
</xsl:template>
```

Each form is configured to submit a `config` request (this is the role of the hidden `input` element in each form).

The drop-down list of themes is generated using an `<imu:select type="themes">` element. When the HTML is generated, this element is replaced with an HTML `<select>` element containing all of the available themes:

```
<select name="theme">
    <option value="london" selected="selected">London</option>
    <option value="new-york">New York</option>
</select>
```

When the user clicks on the `Go` button to the right of the list of themes the form is submitted with a `theme` parameter, such as:

```
http://.../imu.php?request=config&theme=new-york
```

When the request is processed the `PreferredTheme` column of the users record is updated.

The language form operates similarly. The list of languages is generated by a common template, `common-language`:

```
<xsl:template name="london-common-language">
    <select name="lang">
        <option value="en">
            <xsl:if test="$imu-lang = 'en'">
                <xsl:attribute name="selected">
                    <xsl:text>selected</xsl:text>
                </xsl:attribute>
            </xsl:if>
            <xsl:text>English</xsl:text>
        </option>
        <option value="fr">
            <xsl:if test="$imu-lang = 'fr'">
                <xsl:attribute name="selected">
                    <xsl:text>selected</xsl:text>
                </xsl:attribute>
            </xsl:if>
            <xsl:text>Français</xsl:text>
        </option>
        <option value="ar">
            <xsl:if test="$imu-lang = 'ar'">
                <xsl:attribute name="selected">
                    <xsl:text>selected</xsl:text>
                </xsl:attribute>
            </xsl:if>
            <xsl:text>ا ل ع ر ب ي ة</xsl:text>
         </option>
     </select>
</xsl:template>
```

When the user clicks on the `Go` button to the right of the list of languages the form is submitted with a `lang` parameter, such as:

```
http://.../imu.php?request=config&lang=fr
```

When the request is processed the `PreferredLanguage` column of the users record is updated.

The `config` request does not generate any output. Instead it redirects the browser back to the page it came from.

When JavaScript is enabled (page 75) the standard IMu Web JavaScript installs events handlers for both of the drop-down lists so that the form is submitted as soon as a value in the list is chosen. This makes the `Go` buttons redundant so the JavaScript hides them.

If the Go buttons are visible it is a pretty good indication that JavaScript is not enabled (or that the JavaScript has crashed before hiding the buttons).

SECTION 10

Collections

Collections are like shopping carts. They allow the user to create one of more sets of records that they can see and use later.

The information for a collection is stored in EMu's ewebgroups table. This table is similar to the egroups table used by the EMu client. Each ewebgroups record contains the name of a collection and the set of records included in the collection. Unlike EMu client groups, IMu Web collections can contain records from several tables. For this reason ewebgroups records the name of each module along with its irn for each member of the collection.

When a user first requests an IMu Web page - see User Preferences (page 67)- an empty collection called My Collection is automatically created. When the user visits any of the browse or display pages they can add records to this collection by selecting the checkbox associated with the record. The checkboxes are added to these pages using an XSLT template defined in `common.xsl`:

```
<xsl:template name="london-common-checkbox">
    <xsl:param name="suffix"/>
    <xsl:param name="module"/>
    <xsl:param name="key"/>

    <input type="hidden">
        <xsl:attribute name="name">
            <xsl:text>module</xsl:text>
            <xsl:value-of select="$suffix"/>
        </xsl:attribute>
        <xsl:attribute name="value">
            <xsl:value-of select="$module"/>
        </xsl:attribute>
    </input>
    <input type="hidden">
        <xsl:attribute name="name">
            <xsl:text>key</xsl:text>
            <xsl:value-of select="$suffix"/>
        </xsl:attribute>
        <xsl:attribute name="value">
            <xsl:value-of select="$key"/>
        </xsl:attribute>
    </input>
    <input class="group-checkbox" type="checkbox">
        <xsl:attribute name="name">
            <xsl:text>state</xsl:text>
            <xsl:value-of select="$suffix"/>
        </xsl:attribute>
        <xsl:attribute name="module">
            <xsl:value-of select="$module"/>
        </xsl:attribute>
        <xsl:attribute name="key">
            <xsl:value-of select="$key"/>
        </xsl:attribute>
    </input>
</xsl:template>
```

The template takes three parameters, a suffix, a module and a key.

The suffix is the number of the checkbox in the form. As the name implies it is appended to the name of some of the input elements in the form. This is needed because the set of parameters generated must be distinct for each checkbox when the form is submitted without JavaScript.

The module and key parameters are simply the name of the module and the irn (key) of the record associated with the checkbox. Suppose this template was called as follows:

```
<xsl:call-template name="1ondon-common-checkbox">
    <xsl:with-param name="suffix">
       <xsl:text>1<xsl:text>
    </xsl:with-param>
    <xsl:with-param name="module">
        <xsl:text>eparties</xsl:text>
    </xsl:with-param>
    <xsl:with-param name="key">
        <xsl:text>7</xsl:text>
    </xsl:with-param>
</xsl:call-template>
```

The HTML generated would contain three form `<input>` elements, two hidden and one checkbox:

```
<input type="hidden" name="module1" value="eparties"/>
<input type="hidden" name="key1" value="7"/>
<input type="checkbox" name="state1" module="eparties" key="7"/>
```

All of this seems incredibly complicated just to add a checkbox to a page but the checkboxes must be able to operate both with JavaScript and as part of a form submitted without using JavaScript.

If JavaScript is not enabled, the user can select as many records on the page as they want and then click the Update button at the bottom of the page. This submits an IMu Web group request with an action parameter of update, such as:

```
http://...&action=update&module1=eparties&key1=7&state1=on&...
```

This updates the collection, adding entries for records where the state is set to `on` (such as in this example) and removing records where the state is not specified or is set to `off`.

If the user has JavaScript enabled, the record is updated in a slightly different way. The JavaScript installs an event handler on each checkbox. When the checkbox is clicked the event handler extracts the `module` and `key` attributes from the checkbox element and submits a different `group` request using AJAX. The request sets the `action` parameter to `add`, such as:

```
http://.../imu.php?request=group&action=add&module=eparties&key=7
```

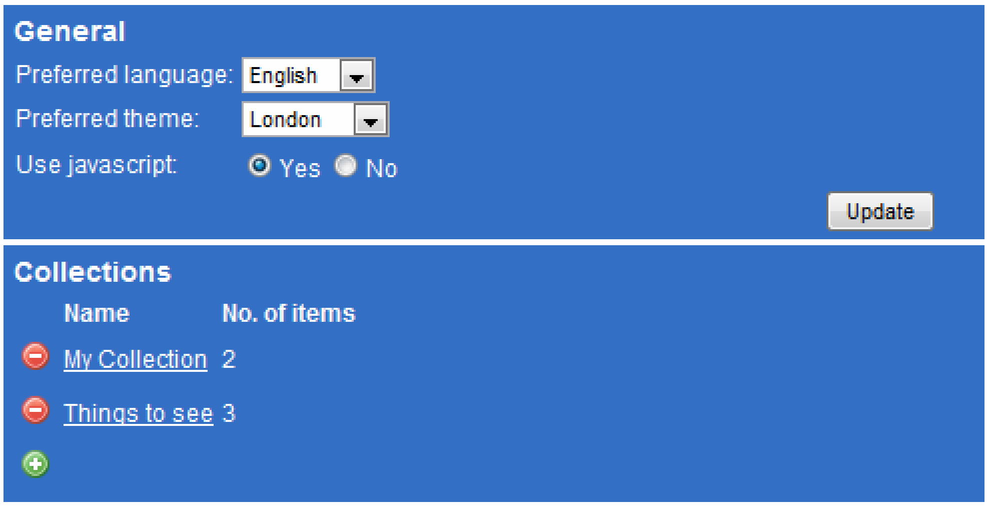
This causes the Parties record with irn 7 to be added to the user's default collection.

The user can rename a collection, create other collections and see the contents of any collection from the My Museum page.

SECTION 11

My Museum

IMu Web includes a My Museum page. The page is divided into two sections. The General section allows the user to update their preferences. The Collections section allows them to manage their collections:


General

Preferred language: English

Preferred theme: London

Use javascript: [✓ Yes] No

Collections

	Name	No. of items
⊖	My Collection	2
⊖	Things to see	3
⊕		

The page is generated by a request of the form:

```
http://web-server/path/imu.php?request=museum
```

To build the General section, the `museum` request retrieves the information from the user's ewebusers record - see *User Preferences* (page 67). As with the other IMu Web pages, the request first generates XML and then transforms it into XHMTL using XSLT. The XML used to build the General section is a follows:

```
<?xml version="1.0"?>
Museum Artifacts
  …
  <irn>86</irn>
  <IPAddress>172.16.57.34</IPAddress>
  <PreferredLanguage>en</PreferredLanguage>
  <PreferredTheme>london</PreferredTheme>
  <UseJavascript>yes</UseJavascript>
  …
</museum>
```

The `museum.xsl` stylesheet used to transform this XML builds a simple form which displays this information and allows the user to change it:

```
<xsl:template name="museum-general">
  <div class="museum-area" id="museum-general">
    <h1><imu:string id="museum-general"/></h1>
    <form>
      <input type="hidden" name="request" value="config"/>
      <table>
        <tr>
          <td>
            <imu:string id="museum-language"/>
            <xsl:text>:</xsl:text>
          </td>
          <td>
            <xsl:call-template name="london-common-language"/>
          </td>
        </tr>
        <tr>
          <td>
            <imu:string id="museum-theme"/>
            <xsl:text>:</xsl:text>
          </td>
          <td>
            <imu:select type="themes" name="theme"/>
          </td>
        </tr>
        <tr>
          <td>
            <imu:string id="museum-javascript"/>
            <xsl:text>:</xsl:text>
          </td>
          <td>
            <input type="radio" name="javascript" value="yes">
              <xsl:if test="UseJavascript = 'yes'">
                <xsl:attribute name="checked">
                  <xsl:text>checked</xsl:text>
                </xsl:attribute>
              </xsl:if>
              <imu:string id="common-yes"/>
            </input>
            <input type="radio" name="javascript" value="no">
              <xsl:if test="UseJavascript != 'yes'">
                <xsl:attribute name="checked">
                  <xsl:text>checked</xsl:text>
                </xsl:attribute>
              </xsl:if>
              <imu:string id="common-no"/>
            </input>
          </td>
        </tr>
      </table>
      <div id="museum-update">
        <imu:submit value="common-update"/>
      </div>
    </form>
  </div>
</xsl:template>
```

Notice that the theme and language drop-down lists are built in the way described in *User Preferences* (page 67).

When this form is submitted it generates a config request, the same request described in *User Preferences* (page 67). For example:

```
http://.../imu.php?request=config&theme=london&lang=fr&javascript=
yes
```

This request updates the user's ewebusers record and refreshes the cookie in the browser.

The Collections section displays the set of collections created by the user. The XML used to build the Collections section is as follows:

```
<?xml version="1.0"?>
Museum Artifacts
  …
  <count>2</count>
  <groups>
    <group>
      <irn>93</irn>
      <GroupName>My Collection</GroupName>
      <count>2</count>
      <entries>
        <entry>
          <module>enarratives</module>
          <key>120002</key>
        </entry>
        <entry>
          <module>ecatalogue</module>
          <key>115686</key>
        </entry>
      </entries>
      <default>no</default>
    </group>
    <group>
      <irn>94</irn>
      <GroupName>Things to see</GroupName>
      <count>3</count>
      <entries>
        <entry>
          <module>enarratives</module>
          <key>1000142</key>
        </entry>
        <entry>
          <module>enarratives</module>
          <key>1000069</key>
        </entry>
        <entry>
          <module>ecatalogue</module>
          <key>120033</key>
        </entry>
      </entries>
      <default>yes</default>
    </group>
  </groups>
  …
</museum>
```

The XSLT which transforms this XML is straightforward. The list of collections is shown in a table. For each collection it includes the collection name and the number of entries in the collection:

```
<xsl:template match="groups/group">
    <tr>
        <td class="museum-action">
            <xsl:if test="count(../group) &gt; 1">
                <a>
                     <xsl:attribute name="href">
                        <xsl:value-of select="$imu-script"/>
                        <xsl:text>?request=group</xsl:text>
                        <xsl:text>&amp;action=delete</xsl:text>
                        <xsl:text>&amp;irn=</xsl:text>
                        <xsl:value-of select="irn"/>
                    </xsl:attribute>
                    <imu:img src="delete.png"/>
                </a>
            </xsl:if>
        </td>
        <td class="museum-name">
            <a>
                <xsl:attribute name="href">
                    <xsl:value-of select="$imu-script"/>
                    <xsl:text>?request=collection</xsl:text>
                    <xsl:text>&amp;irn=</xsl:text>
                    <xsl:value-of select="irn"/>
                </xsl:attribute>
                <xsl:value-of select="GroupName"/>
            </a>
        </td>
        <td class="museum-count">
            <xsl:value-of select="count"/>
        </td>
    </tr>
</xsl:template>
```

The name is marked up as a link. Clicking on a link submits a request such as:

```
http://.../imu.php?request=collection&irn=94
```

This request displays the details for this collection. This is described below.

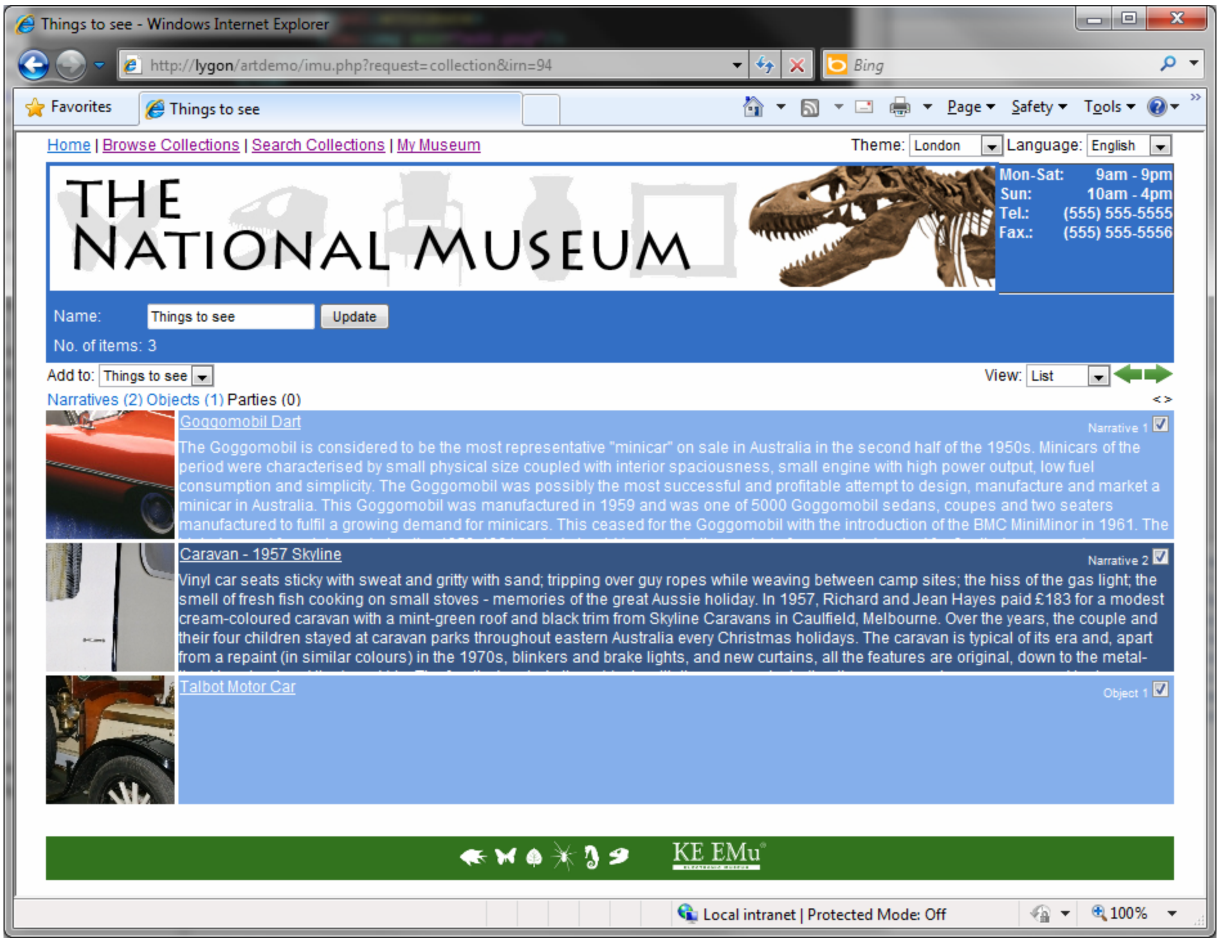
Associated with each collection is an image (in this case `delete.png`). This image is marked up as a link. Clicking on the link submits a request such as:

```
http://.../imu.php?request=group&action=delete&irn=94
```

This request deletes the collection.

Collection Details

When a user clicks on a collection name an IMu Web collection request is generated. This displays details of the collection:



Things to see - Windows Internet Explorer

http://lygon/artdemo/imu.php?request=collection&irn=94

Home | Browse Collections | Search Collections | My Museum

Theme: London Language: English

THE NATIONAL MUSEUM

Mon-Sat: 9am - 9pm
Sun: 10am - 4pm
Tel.: (555) 555-5555
Fax.: (555) 555-5556

Name: Things to see Update

No. of items: 3

Add to: Things to see View: List

Narratives (2) Objects (1) Parties (0)

Goggomobil Dart — Narrative 1

The Goggomobil is considered to be the most representative "minicar" on sale in Australia in the second half of the 1950s. Minicars of the period were characterised by small physical size coupled with interior spaciousness, small engine with high power output, low fuel consumption and simplicity. The Goggomobil was possibly the most successful and profitable attempt to design, manufacture and market a minicar in Australia. This Goggomobil was manufactured in 1959 and was one of 5000 Goggomobil sedans, coupes and two seaters manufactured to fulfil a growing demand for minicars. This ceased for the Goggomobil with the introduction of the BMC MiniMinor in 1961. The

Caravan - 1957 Skyline — Narrative 2

Vinyl car seats sticky with sweat and gritty with sand; tripping over guy ropes while weaving between camp sites; the hiss of the gas light; the smell of fresh fish cooking on small stoves - memories of the great Aussie holiday. In 1957, Richard and Jean Hayes paid £183 for a modest cream-coloured caravan with a mint-green roof and black trim from Skyline Caravans in Caulfield, Melbourne. Over the years, the couple and their four children stayed at caravan parks throughout eastern Australia every Christmas holidays. The caravan is typical of its era and, apart from a repaint (in similar colours) in the 1970s, blinkers and brake lights, and new curtains, all the features are original, down to the metal-

Talbot Motor Car — Object 1

KE EMu

Local intranet | Protected Mode: Off 100%

The collection request generates XML as follows:

```
<?xml version="1.0"?>
<collection>
  <irn>94</irn>
  <name>Things to see</name>
  <count>3</count>
  <entries>
    <entry>
      <module>enarratives</module>
      <key>1000142</key>
    </entry>
    <entry>
      <module>enarratives</module>
      <key>1000069</key>
    </entry>
    <entry>
      <module>ecatalogue</module>
      <key>120033</key>
    </entry>
  </entries>
</collection>
```

The processing of this XML is straightforward. The XSLT stylesheet creates HTML that displays the name of the collection and the number of items in the collection. The name is displayed in a text box. This allows the user to change the name and then update the collection record by clicking the Update button. This submits a group request with an action of rename, such as:

```
http://...&action=rename&irn=94&name=Stuff+to+see
```

If JavaScript is not enabled, the number of records is marked up as a link. Clicking the link displays the first page of records in the collection, in exactly the same way as the results of a search are displayed.

If JavaScript is enabled, the first page of records in the collection is automatically displayed below the collection details (as shown in the screenshot above). This is also identical with the way search results are displayed below the search form when JavaScript is enabled.

SECTION 12

Deployment

Deploying IMu Web to a client is relatively straightforward. Much of the setup is similar to that described in *Development Setup* (page 5) and you should ensure that you have read and understood that section before trying to deploy IMu Web on a client site.

Client back-end and IMu Server

The client's back-end EMu environment must be upgraded to run EMu version 4.0.02 or later. The IMu server must be configured in the same way as described in *Development Setup* (page 5).

Bundling the Development Environment

To deploy IMu Web pages all that is required is to copy the `lib` and `web` directories used in the development environment to the client's web server machine.

The simplest way to copy the files is to first create a "tar ball" on the development machine.

A tar ball is an archive of several files bundled using the Unix `tar` command and compressed using the Unix `gzip` command.

For example, suppose we have developed a set of pages in the IMu `eqag` environment on a development machine. To create the tar ball:

```
development-server$ client eqag
development-server$ tar cvf - lib web | gzip >deploy.tgz
```

This tar ball (in this case `deploy.tgz`) must then be transferred to the client's web server machine.

Installing the Bundle

Once the tar ball has been transferred to the client’s development environment the files must be extracted and placed somewhere where the web server can get access to them. There are many ways that this can be achieved and the method you use will mainly depend on constraints imposed by the administrators of the web server.

Ideally the setup on the client’s web server will be the same as on the development machine as described in *Development Setup* (page 5) and involves setting up an `imu` user account and placing the `imu` user in an `imuadmin` group. This setup is preferred as it provides some additional security.

However, some site administrators may not allow a new user and group to be created. In this case the simplest thing to do is to create a directory owned by the user running the web server.

Once this has been done then the bundled files simply need to be extracted:

```
client-web-server$ mkdir eqag
client-web-server$ cd eqag
client-web-server$ gunzip –c deploy.tgz | tar xvf -
```

This will create the `lib` and `web` directories under the `eqag` directory.

Client's Web Server Setup

Once the IMu Web files have been installed on the client's web server machine, the client's web server machine must be setup. This setup must be done in the same way as described in *Development Setup* (page 5). This requires ensuring that the web server is configured to run PHP 5.0 or later, that the required PHP extensions (ctype, dom, json and xsl) are all enabled and that a suitable mapping between an IMU Web URL and the actual base directory is created.

Index